CENTRE DE RECHERCHE SUR LES TRANSPORTS
(C.R.T.)
CENTRE FOR RESEARCH ON TRANSPORTATION

UNIVERSITÉ DE MONTRÉAL
ÉCOLE DES HAUTES ÉTUDES COMMERCIALES
ÉCOLE POLYTECHNIQUE

MÉMOIRE D'ÉTUDIANT

CENTRE DE RECHERCHE SUR LES TRANSPORTS
UNIVERSITÉ DE MONTRÉAL
C.P. 6128, SUCCURSALE A.
MONTRÉAL
CANADA H3C 3J7

TÉLÉPHONE: (514) 343-7575
TÉLEX: 05-24146 BIBPOLYTEC
EMAIL: crt@crt.umontreal.ca
TÉLÉCOPIEUR (FAX): (514) 343-7121

CRT-761

Implantations parallèles d'un algorithme de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage

par

Bernard Gendron

Mémoire de Maîtrise ès Sciences en informatique, département d'informatique et de recherche opérationnelle, Faculté des arts et des sciences, dirigé par Teodor Gabriel Crainic.

Centre de recherche sur les transports — Publication #761

Février 1991

91-02934/

Dépôt légal – 1er trimestre 1991

SOMMAIRE

Ce mémoire s'inscrit dans le cadre des multiples études réalisées ces dernières années, sur les applications du parallélisme à des problèmes d'optimisation en recherche opérationnelle.

Dans cette optique, nous nous sommes intéressés au problème de localisation avec équilibrage. L'étude de ce problème donne lieu à la formulation d'un modèle de programmation mixte, contenant des variables entières de localisation, pour laquelle un algorithme de séparation et évaluation progressive a été développé. Cet algorithme utilise une relaxation du modèle, appelée relaxation faible, pour calculer une borne inférieure à chaque noeud de l'arbre de séparation et évaluation progressive. L'exploration de l'arbre s'effectue par une recherche en profondeur.

Pour explorer les possibilités de parallélisme offertes par cet algorithme, nous disposons d'un ensemble de processeurs, appelés Transputers, qui se greffent à un micro-ordinateur et utilisent les ressources offertes par ce dernier. Les Transputers peuvent être reliés entre eux selon plusieurs structures.

Le but de ce mémoire est de réaliser des implantations parallèles, sur des Transputers, de l'algorithme de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage.

Nous identifions deux sources principales de parallélisme dans cet algorithme:

- l'exploration parallèle de l'arbre;
- le calcul parallèle de la relaxation faible.

L'exploration parallèle de l'arbre se fait au moyen d'une approche maître-esclave, dans laquelle un seul processeur, appelé maître, génère les noeuds de l'arbre. Ces derniers sont évalués en parallèle par l'ensemble des processeurs.

Des tests effectués sur l'exploration parallèle de l'arbre, se dégagent les conclusions suivantes:

- Une implantation parallèle qui explore en priorité des régions profondes de l'arbre fournit des solutions généralement similaires à celles obtenues par l'implantation séquentielle (ceci, lorsque l'arbre n'est pas complètement exploré).

- Une telle implantation est réalisée de manière efficace, par une phase préliminaire d'exploration séquentielle qui se poursuit jusqu'à l'identification d'une première solution réalisable, de même que par un tri de la liste des problèmes générés mais non-examinés, selon la profondeur.

- L'accélération augmente constamment en fonction du nombre de processeurs, mais de façon sous-linéaire.

- Pour un nombre de processeurs donné, les accélérations sont meilleures pour les problèmes de grande taille.

- Les processeurs peuvent être reliés entre eux de manière efficace au moyen de structures d'arbres. Les arbres les plus performants sont ceux pour lesquels chaque noeud a le plus grand nombre de fils possible.

- Pour des données provenant d'une application réelle, l'exploration parallèle à 16 processeurs prend 45 minutes, alors que l'exploration séquentielle requiert 3 heures.

Le calcul parallèle de la relaxation faible repose sur la résolution de plusieurs problèmes de flot à coût minimum mutuellement indépendants. Les processeurs se partagent entre eux les problèmes à résoudre de telle sorte que chacun en ait à peu près le même nombre.

Sur des données provenant d'une application réelle, les problèmes à résoudre sont de tailles très différentes, ce qui entraîne un mauvais balancement de la charge entre les processeurs et, conséquemment, une perte d'efficacité dans les performances de l'implantation parallèle.

Plusieurs avenues de recherche s'ouvrent à la suite de ce travail. Citons, entre autres:

- réaliser une implantation qui assure un meilleur balancement de la charge entre les processeurs pour le calcul parallèle de la relaxation faible;

- fondre l'exploration parallèle de l'arbre et le calcul parallèle de la relaxation faible en une seule implantation; une telle implantation peut être désignée par le terme "hiérarchie d'esclaves", les esclaves s'occupant d'évaluer des noeuds de l'arbre ayant eux-mêmes sous leurs ordres des esclaves chargés du calcul parallèle de la relaxation faible;

- paralléliser la phase préliminaire d'exploration séquentielle;

- identifier et tester des critères auxiliaires, s'ajoutant à la profondeur, pour le tri de la liste des problèmes générés mais non-examinés;

- calculer la borne inférieure au moyen d'une méthode de résolution basée sur le dual du modèle; étudier ses effets sur l'exploration parallèle et implanter son calcul en parallèle.

Mots clés: parallélisme, Transputer, problème de localisation avec équilibrage, séparation et évaluation progressive, relaxation faible, approche maître-esclave.

TABLE DES MATIÈRES

SOMMAIRE iii

TABLE DES MATIÈRES vi

LISTE DES TABLEAUX ix

LISTE DES FIGURES xi

INTRODUCTION 1

CHAPITRE 1 INTRODUCTION AU PARALLÉLISME 3

1.1 Modèle de von Neumann 4

1.2 Classification de Flynn 5

1.3 Caractéristiques des ordinateurs parallèles 7

1.3.1 Topologies des réseaux d'interconnexion 8

1.3.1.1 Graphe complet 9

1.3.1.2 Arbre 9

1.3.1.3 Maille 12

1.3.1.4 Hypercube 15

1.4 Programmation parallèle 15

1.4.1 Interaction algorithme-ordinateur 15

1.4.2 Conception d'algorithmes parallèles 16

1.4.3 Langages 17

1.5 Alternatives au modèle de von Neumann 18

1.5.1 Modèle "dataflow" 18

1.5.2 Modèle fonctionnel 20

CHAPITRE 2 ENVIRONNEMENT PARALLÈLE SUR TRANSPUTERS .. 21

2.1 Modèle 21

2.2 Matériel 22

2.3 Logiciels 23

2.3.1 Parallel Fortran 23

2.3.1.1 Programmation directe 24

2.3.1.2 "Processor farms" 24

2.3.2 Express 25
2.3.2.1 Approche "host-node" 25
2.3.2.2 Approche Cubix 26
2.3.2.3 Système de communication 27

CHAPITRE 3 LE PROBLÈME DE LOCALISATION AVEC ÉQUILIBRAGE 28
3.1 Description du problème 28
3.2 Modélisation 30
3.3 Approches de résolution 35

CHAPITRE 4 ALGORITHME DE SÉPARATION ET ÉVALUATION PROGRESSIVE 36
4.1 Algorithme général de séparation et évaluation progressive 36
4.2 Algorithme de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage 40
4.2.1 Relaxation faible 41
4.2.2 Borne supérieure 45
4.2.3 Stratégie de parcours de l'arbre 45
4.2.4 Règles de branchement 47
4.2.5 Algorithme 50
4.2.6 Implantation 52

CHAPITRE 5 IMPLANTATIONS PARALLÈLES 54
5.1 Algorithmes parallèles de séparation et évaluation progressive 54
5.1.1 Approche collégiale 56
5.1.2 Approche maître-esclave 58
5.2 Implantations parallèles de l'algorithme de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage 58
5.2.1 Exploration parallèle de l'arbre 59
5.2.1.1 Choix d'une approche 59
5.2.1.2 Choix de modèles de programmation 60
5.2.1.3 Algorithme 64
5.2.1.4 Implantations 68
5.2.2 Calcul parallèle de la relaxation faible 69
5.2.3 Conclusion 71

CHAPITRE 6 RÉSULTATS NUMÉRIQUES 73
6.1 Problèmes testés 74
6.2 Mesures de performance 74
6.3 Problèmes générés aléatoirement 75
6.3.1 Implantation séquentielle 76
6.3.2 Exploration parallèle de l'arbre 77
6.3.2.1 Parallel Fortran 77
6.3.2.1.1 Durée de l'exploration séquentielle 77
6.3.2.1.2 Variation du nombre de processeurs 82
6.3.2.1.3 Efficacité du tri selon la profondeur 85
6.3.2.2 Express 86
6.3.2.2.1 Variation du nombre de processeurs 86
6.3.2.2.2 Efficacité du tri selon la profondeur 94
6.3.2.2.3 Topologie du réseau d'interconnexion 95
6.3.3 Calcul parallèle de la relaxation faible 99
6.4 Application réelle 100
6.5 Conclusion 103

CONCLUSION 105

BIBLIOGRAPHIE 108

REMERCIEMENTS xii

LISTE DES TABLEAUX

Tableau I	Comparaison de la programmation directe et de la technique "processor farms", sur un exemple	62
Tableau II	Performances de l'implantation séquentielle	76
Tableau III	Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC*=4, *N*=6)	78
Tableau IV	Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC*=4, *N*=9)	79
Tableau V	Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC*=4, *N*=12)	79
Tableau VI	Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC*=4, *N*=15)	80
Tableau VII	Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC*=4, *N*=18)	80
Tableau VIII	Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC*=4, *N*=21)	81
Tableau IX	Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC*=4, *N*=24)	81
Tableau X	Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC*=2, *N*=24)	83
Tableau XI	Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC*=3, *N*=24)	83
Tableau XII	Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC*=4, *N*=24)	84
Tableau XIII	Performances de l'exploration parallèle sans tri selon la profondeur, avec Parallel Fortran (*NPROC*=4, *N*=24)	85
Tableau XIV	Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire (*NPROC*=2, *N*=24)	88
Tableau XV	Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire (*NPROC*=3, *N*=24)	88

Tableau XVI	Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire (*NPROC*=4, *N*=24)	89
Tableau XVII	Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire (*NPROC*=6, *N*=24)	89
Tableau XVIII	Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire (*NPROC*=8, *N*=24)	90
Tableau XIX	Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire (*NPROC*=10, *N*=24)	90
Tableau XX	Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire (*NPROC*=12, *N*=24)	91
Tableau XXI	Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire (*NPROC*=14, *N*=24)	91
Tableau XXII	Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire (*NPROC*=16, *N*=24)	92
Tableau XXIII	Performances de l'exploration parallèle sans tri selon la profondeur, avec Express sur une topologie d'arbre binaire (*NPROC*=16, *N*=24)	95
Tableau XXIV	Performances de l'exploration parallèle, avec Express sur une topologie d'arbre 3-aire (*NPROC*=16, *N*=24)	98
Tableau XXV	Performances de l'exploration parallèle, avec Express sur une topologie d'arbre "4-3" (*NPROC*=16, *N*=24)	99
Tableau XXVI	Performances du calcul parallèle de la relaxation faible (*NPROC*=4)	100
Tableau XXVII	Performances des implantations sur les données de l'application réelle	102

LISTE DES FIGURES

Figure 1	Ordinateur de type von Neumann	4
Figure 2	Ordinateur SIMD	6
Figure 3	Ordinateur MIMD	7
Figure 4	Graphe complet	10
Figure 5	Arbre k-aire à n niveaux	11
Figure 6	Réseau linéaire	11
Figure 7	Réseau en étoile	11
Figure 8	Maille à deux dimensions	13
Figure 9	Maille enveloppée à deux dimensions	13
Figure 10	Anneau	14
Figure 11	Exemples d'hypercubes	14
Figure 12	Mécanisme de contrôle "dataflow"	19
Figure 13	Mécanisme de contrôle fonctionnel	19
Figure 14	Structure de réseau du problème de localisation avec équilibrage	32
Figure 15	Exemple d'arbre de séparation et évaluation progressive	38
Figure 16	Exemple d'arbre binaire de séparation et évaluation progressive créé par un parcours en profondeur	46
Figure 17	Topologie d'arbre binaire balancé à 16 Transputers	87
Figure 18	Courbes des accélérations pour les problèmes MZ1P1 et MZ4P6	93
Figure 19	Topologie d'arbre 3-aire à 16 Transputers	97
Figure 20	Topologie d'arbre "4-3" à 16 Transputers	97

INTRODUCTION

Ce travail s'inscrit dans le cadre des multiples études réalisées ces dernières années, sur les applications du parallélisme à des problèmes d'optimisation en recherche opérationnelle.

Dans cette optique, nous nous sommes intéressés au problème de localisation avec équilibrage. L'étude de ce problème entraîne la formulation d'un modèle de programmation mixte contenant des variables entières de localisation et des variables continues de flot (Crainic, Dejax et Delorme, 1989). Pour résoudre ce problème, un algorithme de séparation et évaluation progressive (communément appelé "branch-and-bound") a été développé (Crainic, Delorme et Dejax, 1989). Cet algorithme utilise une relaxation du modèle, appelée relaxation faible, pour calculer une borne inférieure à chaque noeud de l'arbre d'exploration des solutions.

Pour explorer les possibilités de parallélisme offertes par cet algorithme, nous disposons d'un ensemble de processeurs, appelés Transputers, qui se greffent à un micro-ordinateur et utilisent les ressources offertes par ce dernier. Chaque Transputer exécute sa propre séquence d'instructions, qui sont programmées à l'aide d'extensions de langages conventionnels. Ces extensions permettent aux Transputers de communiquer entre eux lors de l'exécution en parallèle de leurs programmes.

Le but de cette étude est de réaliser des implantations parallèles, sur des Transputers, de l'algorithme de séparation et évaluation progressive développé pour résoudre le problème de localisation avec équilibrage.

Cet ouvrage s'ouvre par un chapitre d'introduction au parallélisme, qui vise à donner une vue d'ensemble du domaine. Le chapitre 2 décrit les ressources matérielles et logicielles utilisées dans le cadre de ce travail. Le problème de localisation avec équilibrage et le modèle auquel il donne lieu, sont présentés au chapitre 3. Le chapitre 4 rappelle la méthode générale de séparation et évaluation progressive et l'adaptation qui en a été faite pour résoudre le problème de localisation avec équilibrage. Dans le

chapitre 5, nous faisons d'abord un survol de la littérature sur les algorithmes parallèles de séparation et évaluation progressive. Par la suite, nous décrivons les implantations parallèles de l'algorithme de séparation et évaluation progressive développé pour résoudre le problème de localisation avec équilibrage. Les résultats numériques obtenus par ces implantations font l'objet du chapitre 6. En terminant, nous tirons des conclusions générales de cette étude et nous identifions quelques avenues de recherche.

CHAPITRE 1

INTRODUCTION AU PARALLÉLISME

Depuis la création des premiers ordinateurs, dans le milieu des années 40, le besoin de machines de plus en plus rapides n'a pas cessé de croître.

Pour diminuer le temps d'exécution des instructions d'un programme, Emmen (1987) mentionne trois possibilités:

- des composants plus rapides;
- le traitement pipeline;
- le parallélisme.

Le gain obtenu par la première possibilité est limité par une simple loi physique: un signal électronique ne peut voyager plus vite que la vitesse de la lumière. Cette vitesse est presqu'atteinte dans les superordinateurs actuels (Emmen, 1987).

La deuxième possibilité, le traitement pipeline, est à la base des ordinateurs vectoriels. Lorsqu'une instruction peut être exécutée en plusieurs étapes sur un vecteur de données, elle est candidate pour le traitement pipeline. Celui-ci consiste à exécuter une étape sur une composante du vecteur, concurremment à l'exécution d'étapes antérieures sur d'autres composantes du vecteur.

La troisième possibilité, le parallélisme, implique l'utilisation de plusieurs processeurs dans un même ordinateur. C'est cet aspect que nous considérons dans ce chapitre.

Nous étudions d'abord le modèle de von Neumann, sur lequel est basée l'architecture des ordinateurs séquentiels traditionnels. La classification de Flynn (1966, 1972), désormais classique, nous sert de base à la présentation des ordinateurs parallèles. Les caractéristiques de ceux-ci sont ensuite étudiées plus en détail. Les défis posés par

l'introduction des ordinateurs parallèles aux concepteurs d'algorithmes et aux programmeurs sont alors passés en revue. Enfin, nous terminons par une présentation d'alternatives au modèle de von Neumann, alternatives qui ont pour but de faciliter l'exploitation de certaines formes de parallélisme.

1.1 Modèle de von Neumann

Le mode de fonctionnement de la plupart des ordinateurs en usage de nos jours suit encore les principes de base élaborés au milieu des années 40 et attribués à John von Neumann. Une unité de contrôle va chercher une instruction et ses opérandes d'une mémoire centrale et les transmet à un processeur. L'instruction est alors exécutée et le résultat est envoyé à la mémoire centrale.

La figure 1 montre un schéma simplifié d'un ordinateur de type von Neumann.

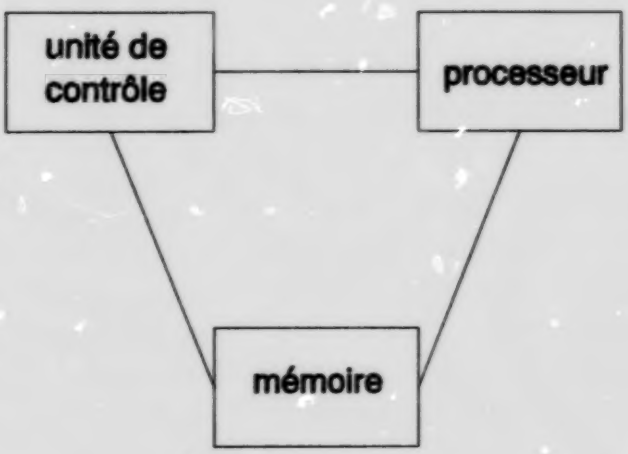

Figure 1. Ordinateur de type von Neumann

Les performances du modèle sont limitées par le mode d'exécution séquentiel des instructions. Les architectes d'ordinateur se sont alors tournés vers d'autres types de machines, en particulier les ordinateurs vectoriels, basés sur le traitement pipeline, et les ordinateurs parallèles.

1.2 Classification de Flynn

La classification des ordinateurs proposée par Flynn (1966, 1972) est basée sur les notions de flot d'instructions et de flot de données. Par flot d'instructions, on entend une séquence d'instructions traitées par une unité de contrôle, alors que le flot de données représente une séquence d'opérandes recueillis en mémoire. Les ordinateurs parallèles sont caractérisés par la multiplicité du flot d'instructions ou du flot de données.

À partir de ces principes, Flynn distingue quatre grandes classes d'ordinateurs:

- la classe SISD ("single instruction stream - single data stream");
- la classe MISD ("multiple instruction stream - single data stream");
- la classe SIMD ("single instruction stream - multiple data stream");
- la classe MIMD ("multiple instruction stream - multiple data stream").

La classe SISD est constituée des ordinateurs qui ne peuvent exécuter qu'une seule instruction à la fois sur une seule donnée. Les ordinateurs séquentiels traditionnels, conçus sur le modèle de von Neumann, appartiennent à cette classe.

La classe MISD est constituée des ordinateurs pouvant exécuter plusieurs instructions à la fois sur une seule donnée. De tels ordinateurs ne peuvent traiter que des problèmes très particuliers (Flynn, 1972; Akl, 1989). Pour cette raison, ils n'ont reçu que peu d'attention des chercheurs.

Les ordinateurs de la classe SIMD exécutent une seule instruction à la fois mais sur plusieurs jeux de données. Les ordinateurs vectoriels, utilisant le traitement pipeline, sont souvent vus comme membres de cette classe (Emmen, 1987). Les ordinateurs parallèles appartenant à cette classe, sont constitués d'une seule unité de contrôle qui dicte les instructions à être exécutées par les processeurs (voir figure 2).

La classe MIMD comprend tous les ordinateurs pouvant exécuter plusieurs instructions à la fois sur des jeux de données différents. Les ordinateurs parallèles de cette classe sont constitués de plusieurs unités de contrôle, chacune ayant généralement sous ses ordres un processeur (voir figure 3). Notons qu'un ordinateur MIMD peut simuler le mode de fonctionnement d'un ordinateur SIMD lorsque les mêmes instructions s'exécutent "en même temps" (de façon synchronisée) sur tous les processeurs.

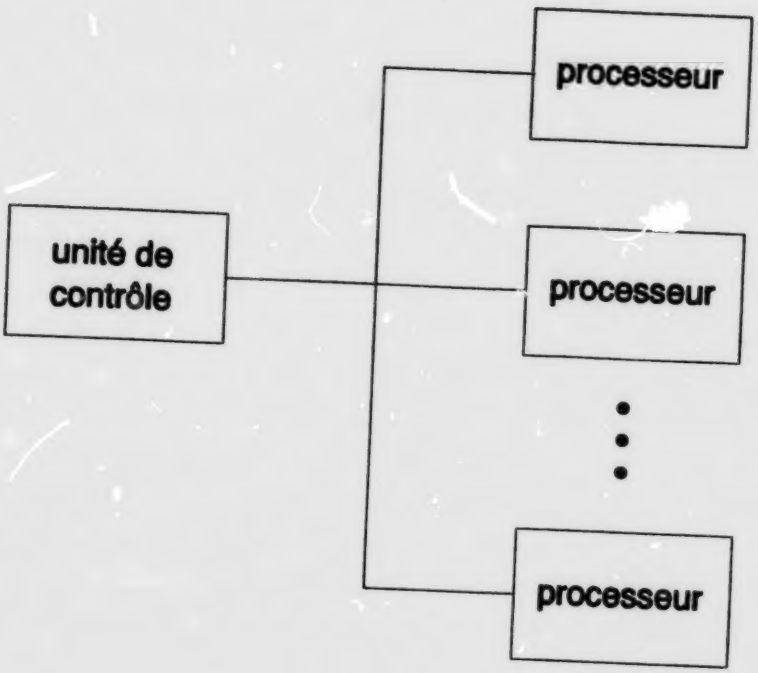

Figure 2. Ordinateur SIMD

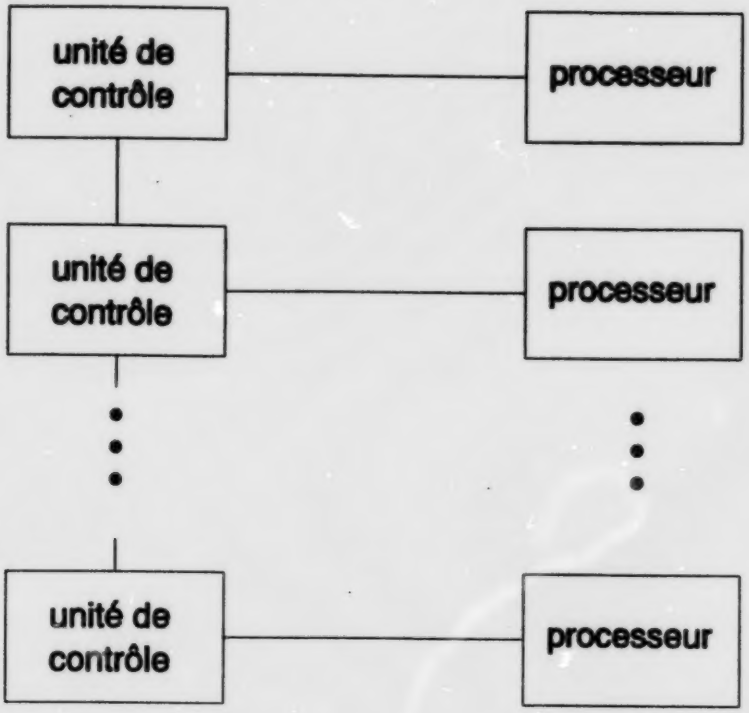

Figure 3. Ordinateur MIMD

Dans la suite du texte, nous nous intéressons uniquement aux ordinateurs parallèles, appartenant aux classes SIMD et MIMD.

1.3 Caractéristiques des ordinateurs parallèles

Bertsekas et Tsitsiklis (1989) identifient quelques paramètres permettant de caractériser les ordinateurs parallèles:

- *Le type et le nombre de processeurs.*
  Certains ordinateurs, appelés **massivement parallèles**, possèdent des milliers de processeurs peu puissants mais permettant d'exploiter efficacement un parallélisme à **grain fin** (par exemple, au niveau des opérations élémentaires). À l'opposé, d'autres ordinateurs sont constitués de peu de processeurs (de l'ordre de 10) et sont appropriés pour exploiter un parallélisme à **gros grain** (par exemple, au niveau des boucles ou des sous-programmes).

- *La présence ou l'absence d'un mécanisme de contrôle global.*
  Les ordinateurs SIMD possèdent un tel mécanisme puisqu'ils sont constitués d'une seule unité de contrôle. Ce n'est pas le cas des ordinateurs MIMD.

- *Le mode d'opération: synchrone ou asynchrone.*
  Certains ordinateurs possèdent une horloge commune à tous les processeurs qui permet de synchroniser les opérations. C'est le cas des ordinateurs SIMD. D'autres ordinateurs fonctionnent en mode asynchrone, mais peuvent simuler un mode d'opération synchrone (Bertsekas et Tsitsiklis, 1989.)

- *Les échanges d'informations entre les processeurs.*
  Pour réaliser les échanges d'informations dans un système multiprocesseur, deux alternatives extrêmes sont possibles: les systèmes multiprocesseurs à **mémoire partagée** et les systèmes multiprocesseurs à **échanges de messages**. Dans le premier cas, tous les processeurs ont accès simultanément à une mémoire commune. Pour éviter les conflits d'accès à une même cellule, la mémoire est souvent divisée en blocs, dont le nombre est comparable à celui des processeurs. Ceux-ci ont accès à tous les blocs à travers un réseau de commutateurs (Athas et Seitz, 1988; Akl, 1989; Bertsekas et Tsitsiklis, 1989). Dans le cas des systèmes à échanges de messages, les processeurs possèdent une mémoire propre et communiquent des informations à travers un **réseau d'interconnexion**.

Dans les systèmes multiprocesseurs à échanges de messages, la topologie du réseau d'interconnexion (c'est-à-dire la façon dont les processeurs sont reliés entre eux) est un aspect crucial pour la communication des informations. Nous étudions quelques exemples de topologies dans la prochaine sous-section.

1.3.1 Topologies des réseaux d'interconnexion

Afin de mesurer les différentes topologies, Bertsekas et Tsitsiklis (1989) introduisent plusieurs paramètres, dont:

- *Le diamètre.*
  C'est la distance maximum entre toutes les paires de processeurs du réseau. La distance est le nombre minimum de liens à traverser pour aller d'un processeur vers un autre.

- *La connectivité.*
  C'est le nombre minimum d'arcs à éliminer pour que le réseau devienne non-connexe.

- *La flexibilité.*
  C'est la possiblité d'obtenir une autre topologie à partir de la topologie donnée en se restreignant à des sous-ensembles des processeurs et des arcs.

Nous avons choisi de discuter des propriétés de quatre grandes familles de topologies: le graphe complet, l'arbre, la maille et l'hypercube. La grande majorité des topologies traitées dans la littérature peuvent être incluses dans ces quatre familles. Nous supposons que chaque topologie possède p processeurs.

1.3.1.1 Graphe complet

Dans cette topologie, chaque processeur est relié à tous les autres (voir figure 4). Dans l'ensemble des topologies à p processeurs, son diamètre (=1) est minimum et sa connectivité ($= p - 1$) est maximum. Sa flexibilité est idéale puisqu'on peut obtenir n'importe quelle autre topologie à partir du graphe complet.

Malheureusement, cette topologie n'est pas très réaliste pour un nombre élevé de processeurs. En effet, elle nécessite un grand nombre de liens ($p(p-1)/2$). Relier les processeurs par un commutateur à barres croisées peut alors être très coûteux, tandis qu'une structure de bus peut amener des délais d'attente imposants (Bertsekas et Tsitsiklis, 1989).

1.3.1.2 Arbre

Un arbre k-aire (dont chaque noeud possède au maximum k fils) à n niveaux, dont la racine se trouve au niveau 0, est illustré à la figure 5. Il permet de relier toutes les paires de processeurs avec un nombre minimum de liens ($p - 1$). Sa connectivité (=1) est minimum dans l'ensemble des topologies et par conséquent, sa flexibilité est très faible. Le diamètre dépend de la structure de l'arbre.

Mentionnons quelques types particuliers d'arbres.

Le réseau linéaire est la topologie la plus simple permettant de relier entre eux p processeurs (voir figure 6). On peut le considérer comme un arbre en posant $k = 1$. Son diamètre ($= p - 1$) est le plus élevé parmi toutes les structures d'arbre et même parmi toutes les topologies à p processeurs.

Le réseau en étoile relie $(p - 1)$ processeurs à un processeur racine (voir figure 7). On peut donc le considérer comme un arbre $(p - 1)$-aire à deux niveaux. Son diamètre (=2) est minimum parmi toutes les structures d'arbre.

L'arbre balancé est un arbre dont toutes les feuilles se trouvent à une distance $n-1$ ou $n-2$ de la racine ($n = \lceil \log_k(p(k-1)+1) \rceil$).

L'arbre complet est un arbre pour lequel chaque niveau i $(0 \le i \le n-1)$ possède exactement k^i noeuds. Pour un arbre complet, on a la relation $p = (k^n - 1)/(k - 1)$.

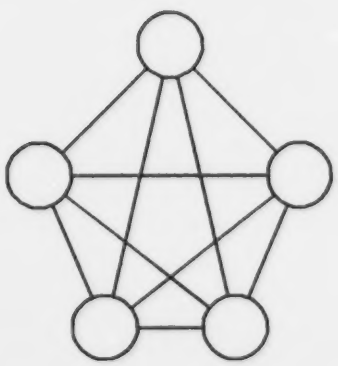

Figure 4. Graphe complet

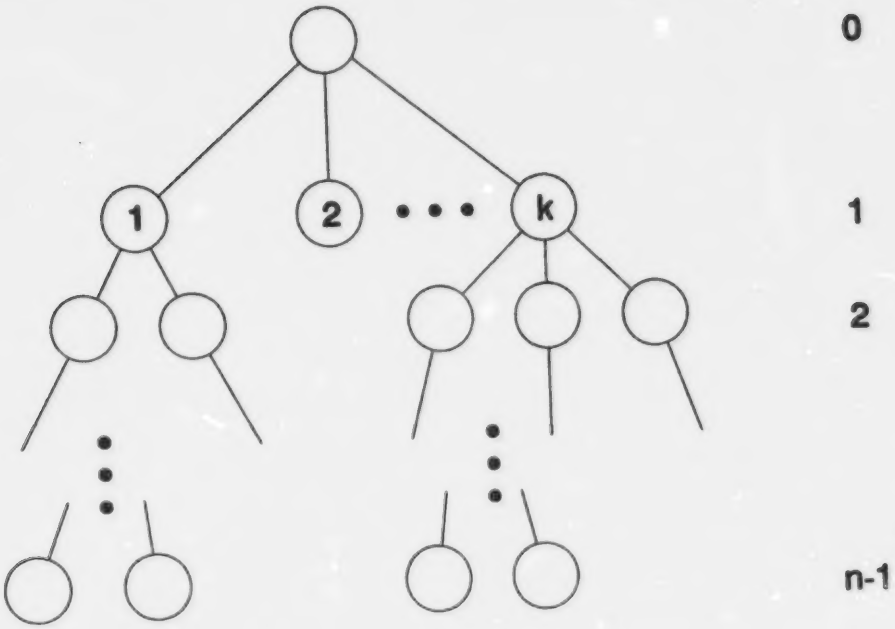

Figure 5. Arbre k-aire à n niveaux

Figure 6. Réseau linéaire

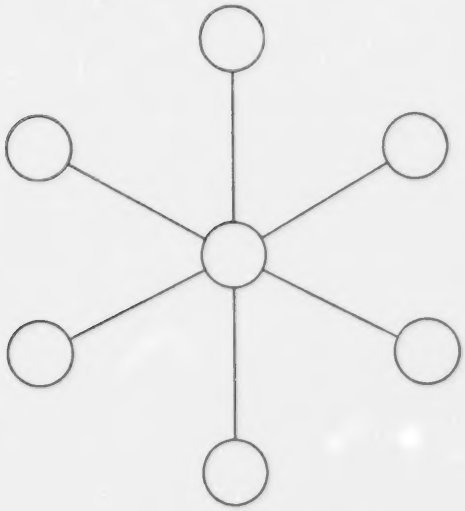

Figure 7. Réseau en étoile

1.3.1.3 Maille

Dans une maille à d dimensions, les processeurs occcupent les points entiers d'un espace à d dimensions et un lien direct relie deux processeurs séparés par une distance unitaire le long d'une dimension (voir figure 8). Formellement, chaque noeud d'une maille à d dimensions avec n_i points le long de la $i^{ème}$ dimension est un d-uplet $(x_1, \ldots, x_d)$ satisfaisant les conditions suivantes:

1) $x_i = 1, \ldots, n_i, \quad i = 1, \ldots, d;$

2) Il y a un lien entre $(x_1, \ldots, x_d)$ et $(x_1', \ldots, x_d')$ si et seulement si il existe k tel que $|x_k - x_k'| = 1$ et $x_j = x_j'$, pour tout $j \neq k$.

Avec cette définition, il est facile d'établir que le diamètre de la maille est $\sum_{i=1}^{d} (n_i - 1)$ et que sa connectivité est d. Sa flexibilité dépend grandement de la structure de la maille, c'est-à-dire du nombre de ses dimensions et du nombre de processeurs par dimension.

Mentionnons quelques types particuliers de mailles.

Le réseau linéaire, déjà vu à la section précédente, peut être considéré comme une maille, en posant $d = 1$.

La maille à deux dimensions est utilisée surtout dans la conception de machines cellulaires, adaptées à la résolution de problèmes matriciels.

La maille symétrique possède le même nombre de composantes dans toutes les dimensions ($n_i = p^{1/d}$).

L'hypercube à d dimensions possède deux composantes le long de chaque dimension. Son importance est telle qu'il fait l'objet de la prochaine section.

Une autre topologie proche de la maille est la **maille enveloppée**. Celle-ci ne diffère de la maille que par l'ajout de liens supplémentaires de type

$((x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_d),\quad (x_1',\ldots,x_{i-1}',n_i,x_{i+1}',\ldots,x_d'))$ (voir figure 9). Par rapport à la maille, le diamètre est diminué de moitié environ (Bertsekas et Tsitsiklis, 1989) et la connectivité est légèrement augmentée.

Un cas particulier de maille enveloppée est l'**anneau**, illustré à la figure 10, obtenu en posant $d = 1$.

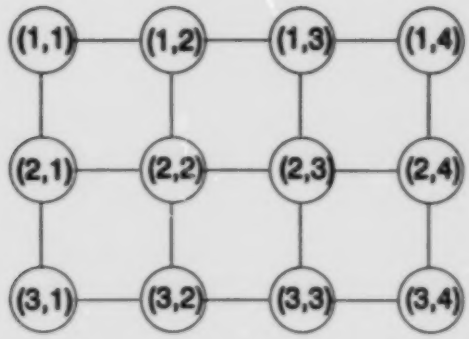

Figure 8. Maille à deux dimensions

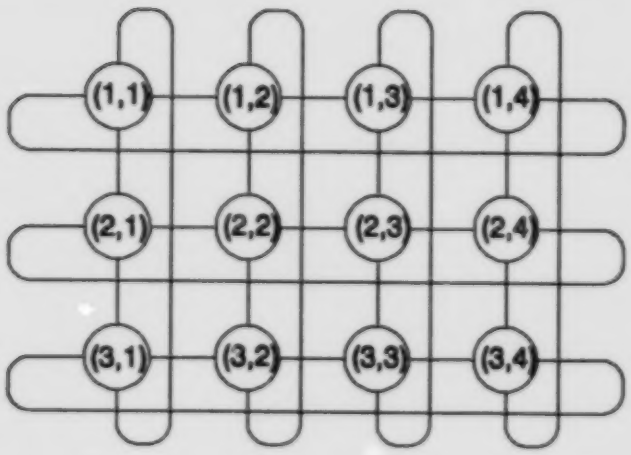

Figure 9. Maille enveloppée à deux dimensions

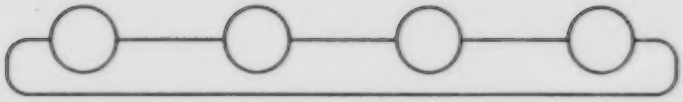

Figure 10. Anneau

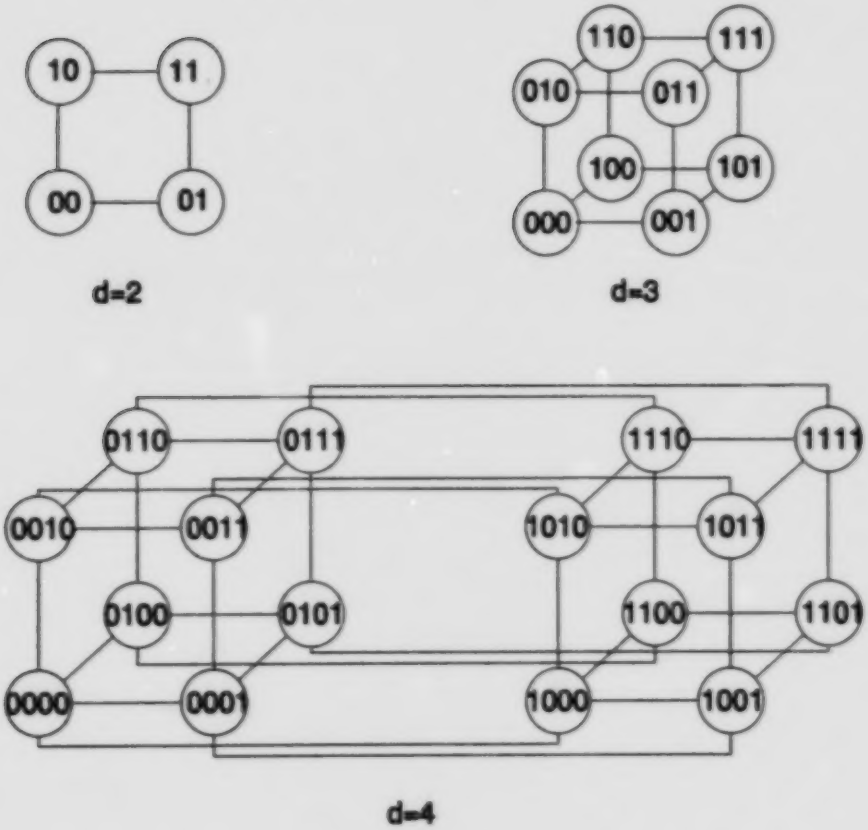

Figure 11. Exemples d'hypercubes

1.3.1.4 Hypercube

Un hypercube à d dimensions possède 2^d processeurs. Chaque processeur correspond à un point d'un espace à d dimensions dont les coordonnées sont des 0 ou des 1. Deux processeurs sont reliés entre eux s'ils ne diffèrent que par une seule coordonnée. La figure 11 illustre des hypercubes à deux, trois et quatre dimensions.

Le diamètre et la connectivité de l'hypercube sont égaux et valent $d = \log p$, où log est le logarithme en base 2. Le nombre total de liens dans un hypercube est égal à $p(\log p)/2$.

Une des propriétés intéressantes de l'hypercube est son excellente flexibilité. Un anneau ayant un nombre pair de processeurs peut être généré à partir d'un hypercube (Bertsekas et Tsitsiklis, 1989). Il est également possible de générer plusieurs mailles, ainsi que des mailles enveloppées (Bertsekas et Tsitsiklis, 1989). Cependant, lorsque $d \geq 3$, il est impossible de générer un arbre binaire complet de $2^d - 1$ processeurs (Bhatt et Ipsen, 1985). Par contre, il est possible de générer un arbre binaire balancé à deux racines possédant 2^d processeurs. Chacune des racines est reliée à l'autre ainsi qu'à la racine d'un arbre binaire balancé (Bhatt et Ipsen, 1985).

1.4 Programmation parallèle

Dans cette section, nous nous intéressons à certaines caractéristiques de la programmation parallèle. Nous étudions d'abord les relations entre la conception d'algorithmes parallèles et le type d'ordinateur utilisé. Nous voyons ensuite quelques aspects propres à la conception d'algorithmes parallèles. Enfin, nous terminons en examinant les types de langages utilisés pour la programmation parallèle.

1.4.1 Interaction algorithme-ordinateur

Bertsekas et Tsitsiklis (1989) mentionnent trois approches possibles pour la conception d'algorithmes parallèles.

La première approche consiste à développer un algorithme parallèle sans s'occuper de son implantation sur un type particulier de machine. Les problèmes spécifiques à cette approche sont de démontrer la convergence de l'algorithme et d'étudier son potentiel d'accélération par rapport à un algorithme séquentiel existant.

La deuxième approche s'intéresse aux détails de l'implantation sur un type particulier de machine. Les problèmes spécifiques à cette approche sont d'étudier la validité de l'algorithme ainsi que sa complexité de temps et de communication.

Dans la troisième approche, l'ordinateur et l'algorithme sont interdépendants, c'est-à-dire que la conception de l'un influence celle de l'autre. Citons l'exemple de composants VLSI conçus pour exécuter un algorithme parallèle précis.

1.4.2 Conception d'algorithmes parallèles

Bertsekas et Tsitsiklis (1989) identifient plusieurs caractéristiques propres à la conception d'algorithmes parallèles, en particulier la **décomposition**, la **communication** et la **synchronisation**.

La décomposition comprend trois étapes principales:
- le **partitionnement** de l'algorithme en sous-tâches;
- l'**allocation** des différentes sous-tâches aux processeurs;
- la détermination de l'**ordre d'exécution** des sous-tâches.

Ces trois étapes doivent être conçues conjointement de façon à assurer un **équilibre de la charge** entre les processeurs.

La conception d'algorithmes parallèles est également influencée par la communication de données et de résultats entre les processeurs. L'objectif principal est alors de minimiser le temps de communication et d'évaluer son impact sur la performance globale de l'algorithme.

Un autre aspect primordial dans la conception d'algorithmes parallèles est la synchronisation. Dans certains algorithmes, appelés **synchrones**, les processeurs doivent attendre à certains points prédéterminés que certains calculs soient complétés ou

que des données soient communiquées. Dans d'autres algorithmes, appelés **asynchrones**, les processeurs n'ont pas à attendre à des points prédéterminés. L'objectif de ces algorithmes est d'éliminer les délais dûs à la synchronisation. Cependant, il peut être difficile d'établir leur validité. En effet, lors de l'exécution d'un tel algorithme, il est possible que des processeurs traitent certaines données, sans attendre la dernière mise à jour de ces données effectuée par d'autres processeurs. Il pourrait en résulter des calculs inutiles, susceptibles d'affecter la convergence de l'algorithme.

Mentionnons en terminant, qu'un autre aspect important de la programmation parallèle s'intéresse à des mesures de performance des algorithmes. Le développement de telles mesures nécessite l'étude d'un modèle théorique d'algorithmique parallèle. Plusieurs modèles ont été proposés (Kindervater et Lenstra, 1988; Akl, 1989; Bertsekas et Tsitsiklis, 1989). Leur étude dépasse largement le cadre de cette introduction.

1.4.3 Langages

Les langages parallèles peuvent être de deux types:

- des extensions de langages séquentiels connus;
- des langages nouveaux, développés pour traiter des applications sur des machines spécifiques.

Les extensions de langages séquentiels sont constituées essentiellement de bibliothèques de fonctions adaptées au traitement parallèle sur une machine donnée. L'avantage d'utiliser de telles extensions est qu'elles permettent de développer des programmes parallèles à partir de programmes séquentiels déjà existants, en laissant intactes certaines parties du code.

Une alternative aux langages parallèles consiste à implanter un algorithme séquentiel sur un ordinateur parallèle et à utiliser un compilateur parallélisant qui détecte certaines formes de parallélisme présentes dans le programme (voir, par exemple, Hilhorst, Braam et Sips, 1987). Le problème majeur avec cette approche est que le concepteur est à la merci du compilateur; seules les formes de parallélisme identifiées par le compilateur seront exploitées.

1.5 Alternatives au modèle de von Neumann

Tous les modèles d'ordinateurs que nous avons étudiés jusqu'ici sont basés sur un fonctionnement par flot d'instructions: la suite des instructions interprétée par une unité de contrôle détermine les actions prises par un processeur. De tels modèles sont, en fait, des généralisations du modèle de von Neumann à des systèmes multiprocesseurs. Ils supposent tous, en effet, une exécution séquentielle des instructions à l'intérieur d'un même processeur. Or, un tel mode de fonctionnement ne permet pas nécessairement d'exploiter pleinement toutes les possibilités de parallélisme. Ainsi, des instructions indépendantes les unes des autres, donc susceptibles d'être exécutées en parallèle, seront plutôt traitées dans un ordre artificiel imposé par le traitement séquentiel.

C'est pourquoi des alternatives à ces modèles ont été proposées afin de mieux exploiter les possibilités de parallélisme. Nous en présentons deux: le modèle "dataflow" et le modèle fonctionnel.

1.5.1 Modèle "dataflow"

L'idée de base de ce modèle est qu'il n'est pas nécessaire d'exécuter les instructions dans l'ordre donné par un programme séquentiel. En fait, tout ordre qui respecte les contraintes de précédence imposées par les données, peut être utilisé.

Ainsi, ce n'est pas une instruction qui permet l'exécution d'une opération, mais celle-ci s'exécute dès que ses opérandes sont disponibles (voir figure 12).

Un programme basé sur le modèle "dataflow" peut être représenté par un graphe dont les noeuds sont des opérations et les arcs indiquent que le résultat à un noeud origine est une donnée utilisée par l'opération au noeud destination. Lors de l'exécution, les arcs sont marqués. Une opération est prête à être exécutée lorsque tous ses arcs incidents sont marqués.

Quelques ordinateurs ont été construits sur le modèle "dataflow" (Emmen, 1987; D'Hollander, 1987). Une alternative moins ambitieuse consiste à utiliser les principes du modèle "dataflow" sur des ordinateurs traditionnels afin d'aider à la parallélisation de programmes écrits dans des langages conventionnels (D'Hollander, 1987).

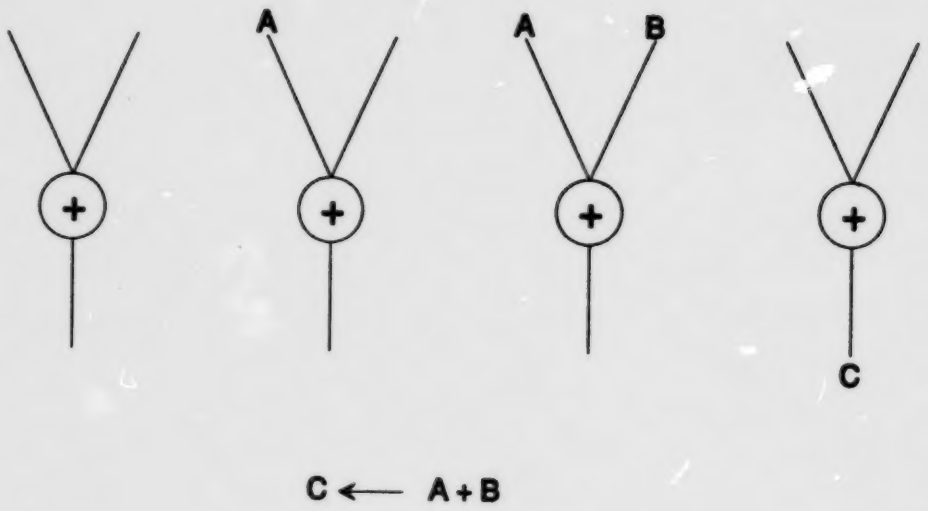

Figure 12. Mécanisme de contrôle "dataflow"

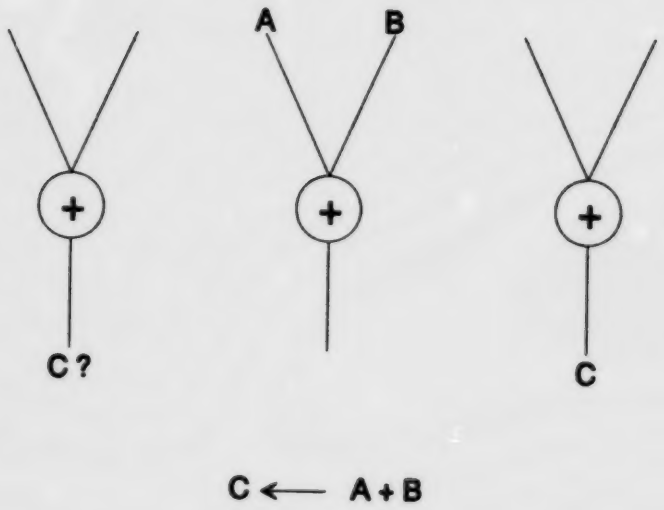

Figure 13. Mécanisme de contrôle fonctionnel

1.5.2 Modèle fonctionnel

Dans ce modèle, appelé aussi modèle de réduction ou modèle "demand flow" (Emmen, 1987), l'exécution d'une opération se fait par la recherche de ses opérandes. Si un opérande est une variable résultant d'une autre opération, les opérandes de cette opération sont recherchés à leur tour. Ce processus se termine lorsque les opérandes sont des constantes connues. Toutes les opérations demandées sont alors exécutées (voir figure 13).

Un programme basé sur le modèle fonctionnel est représenté par un graphe dont les feuilles symbolisent des opérations et des opérandes, et les noeuds internes sont des expressions à être évaluées. Lors de l'exécution, le graphe est réduit progressivement jusqu'à obtenir les résultats du programme. Plusieurs parties du graphe étant indépendantes, elles peuvent être réduites parallèlement (Peyton Jones, 1987).

Quelques machines de réduction parallèle ont été développées (Peyton Jones, 1987). Plusieurs langages fonctionnels ayant déjà été implantés sur des ordinateurs traditionnels, une alternative consiste à exploiter les possibilités de parallélisme offertes par ces langages, sur des architectures conventionnelles (Peyton Jones, 1987).

CHAPITRE 2

ENVIRONNEMENT PARALLÈLE SUR TRANSPUTERS

Ce chapitre est consacré à la description de l'environnement parallèle sur Transputers, utilisé dans le cadre de ce travail. Le modèle abstrait sur lequel est basé le Transputer fait l'objet de la première section. Nous traitons ensuite de l'aspect matériel et nous terminons par une présentation des logiciels pour la programmation sur Transputers.

2.1 Modèle

Un programme parallèle est vu comme une collection de **processus séquentiels communicants** (Hoare, 1985). Les processus communiquent entre eux à travers des **canaux**. Un canal relie exactement un processus à un autre et cette liaison est unidirectionnelle.

Les communications sont **synchronisées**, c'est-à-dire qu'un processus qui envoie un message doit en attendre la réception par un autre processus, avant de poursuivre son exécution. Notons qu'il est toutefois possible d'utiliser un mécanisme de mise en mémoire temporaire des messages afin de permettre à un processus de poursuivre son exécution sans attendre la réception du message qu'il a envoyé.

Dans ce modèle, les processus et leurs canaux sont **statiques**. Ils sont connus avant l'exécution du programme. On ne peut ni ajouter, ni enlever des processus durant l'exécution.

2.2 Matériel

Le Transputer de Inmos est une réalisation matérielle du modèle décrit à la section précédente. C'est un processeur ayant sa propre mémoire et doté de quatre liens, numérotés de 0 à 3. Chaque lien possède deux canaux et permet une communication synchronisée.

Chaque Transputer supporte un système d'exploitation permettant à plusieurs processus de coexister sur un même processeur. Ces processus peuvent communiquer entre eux par des canaux internes spécifiés par des mots en mémoire.

Une réalisation d'un réseau de Transputers est le Quadputer de Microway (1990a), conçu pour fonctionner sur un IBM PC AT ou un compatible. Le Quadputer contient un, deux ou quatre Transputers, dotés chacun d'une mémoire de 1 ou de 4 mégaoctets. Sur un Quadputer, chaque Transputer opère à une fréquence de 20 mégahertz. Ses liens de communication permettent d'acheminer des informations à des vitesses de 10 ou de 20 mégabits/seconde, le choix étant laissé à l'usager.

Deux possibilités s'offrent pour réaliser des réseaux d'interconnexion avec des Quadputers:

- *Les "daughterboards" et les câbles.*
  Les "daughterboards" sont des plaques fixées aux Quadputers. Ils définissent les réseaux d'interconnexion implantés sur les Quadputers. Pour un Quadputer à quatre Transputers, ce réseau est un graphe complet. Pour un Quadputer à deux Transputers, deux liens relient chacun des Transputers. Dans tous les cas, les liens libres peuvent être utilisés pour relier entre eux plusieurs Quadputers au moyen de câbles.

- *Le Linkputer* (Microway, 1990b).
  Le Linkputer est une carte programmable contenant quatre commutateurs à barres croisées, capables de relier entre eux huit Quadputers, ayant chacun quatre Transputers. Chaque commutateur relie entre eux des liens de même numéro.

Peu importe la façon dont les Quadputers sont reliés entre eux, exactement un Transputer, appelé **processeur Racine**, doit être connecté au PC, appelé **processeur Hôte**. Cette liaison permet l'accès aux E/S (entrées/sorties) et le chargement des programmes à partir du processeur Hôte.

2.3 Logiciels

Dans cette section, nous traitons de deux logiciels pour la programmation sur Transputers: Parallel Fortran de 3L (1988) et Express de Parasoft (1990). Le premier est un compilateur Fortran doté de fonctions adaptées à la programmation sur Transputers. Le deuxième est une bibliothèque pour le traitement parallèle dont nous utilisons une version qui fonctionne avec le compilateur Fortran développé par 3L.

2.3.1 Parallel Fortran

Un programme parallèle est vu ici comme une collection de tâches concurrentes. À chaque tâche est associée une région en mémoire, ainsi que des portes d'entrée et des portes de sortie pour la communication avec les autres tâches.

Les portes sont attachées à des adresses de canaux. Les tâches placées sur un même processeur peuvent avoir un nombre quelconque de canaux, alors que celles placées sur des processeurs différents, peuvent communiquer uniquement à travers les liens physiques.

Pour obtenir un programme parallèle exécutable, il faut générer les tâches en compilant les programmes-sources. Il faut également déclarer le matériel (les processeurs et leurs liens physiques) et les tâches (leur emplacement et les canaux les reliant) dans un fichier de configuration. Un configurateur utilise l'information contenue dans ce fichier pour créer le code exécutable.

Il est possible d'exécuter plusieurs **procédures concurrentes** à l'intérieur d'une même tâche. Contrairement aux tâches, les procédures concurrentes partagent les mêmes données en mémoire. Elles peuvent aussi communiquer entre elles par des canaux internes. Pour éviter des conflits d'accès à la mémoire, les procédures concurrentes utilisent des **sémaphores**. Une applicaton intéressante des procédures concurrentes

concerne l'implantation des algorithmes asynchrones (voir section 1.4.2). Dans une telle implantation, des procédures de réception et d'envoi s'exécutent concurremment à une procédure principale, permettant alors un traitement asynchrone des communications.

Deux grands modèles de programmation sont permis par Parallel Fortran: un schéma général, décrit ci-dessous et que nous appellerons programmation directe, et une technique particulière aux applications maître-esclave, appelée technique "processor farms".

2.3.1.1 Programmation directe

Les caractéristiques du modèle de programmation directe sont:

- Les fonctions d'E/S sont accessibles à travers une seule tâche qui doit résider sur le processeur Racine.
- Les communications sont synchronisées et possibles uniquement à travers des canaux.
- Une tâche possédant plusieurs canaux d'entrée a la possibilité d'identifier la provenance d'un message lui étant destiné.
- Une tâche peut vérifier si un message lui est destiné, et dans la négative, poursuivre son exécution.
- Le matériel et les tâches sont entièrement déclarées dans le fichier de configuration.

2.3.1.2 "Processor farms"

Dans ce modèle, une tâche maître sépare le travail en paquets qui sont expédiés à des tâches esclaves. Celles-ci procèdent au traitement des paquets et retournent les résultats à la tâche maître. À son tour, celle-ci répond aux demandes des esclaves, soit en recueillant les résultats, soit en expédiant de nouveaux paquets.

Les caractéristiques de ce modèle sont:

- La tâche maître réside dans le processeur Racine; elle est la seule à pouvoir utiliser des fonctions d'E/S.

- La tâche esclave réside dans tous les Transputers, y compris le processeur Racine.

- La communication se fait au moyen d'une tâche appelée Frouter, résidant sur tous les Transputers. Lorsque la tâche maître expédie un paquet, la tâche Frouter l'achemine vers la première tâche esclave libre.

- Seules les tâches maître et esclave, et la mémoire requise par cette dernière, doivent être déclarées dans le fichier de configuration. Un configurateur spécial permet alors le développement d'un programme pouvant s'exécuter sur un réseau quelconque de Transputers.

2.3.2 Express

Un programme parallèle est vu ici comme un ensemble de programmes séquentiels exécutés concurremment et pouvant communiquer entre eux. À chacun d'eux est associé un processeur sur lequel il s'exécute.

Deux grandes approches de programmation sont permises par Express: l'approche "host-node" et l'approche Cubix. Après une description de ces deux approches, nous voyons par la suite le système de communication qui les soustend.

2.3.2.1 Approche "host-node"

Dans cette approche, un programme s'exécute sur le processeur Hôte et un ou plusieurs programmes s'exécutent sur le réseau de Transputers.

Les caractéristiques de cette approche sont:

- Les fonctions d'E/S sont accessibles uniquement au programme s'exécutant sur le processeur Hôte.

- L'allocation des programmes aux Transputers se fait à partir du programme résidant sur le processeur Hôte.

- Un compilateur supplémentaire est requis pour les programmes s'exécutant sur le processeur Hôte, car le compilateur de 3L, utilisé par Express, n'est adapté qu'au fonctionnement sur les Transputers.

2.3.2.2 Approche Cubix

Dans cette approche, un ou plusieurs programmes s'exécutent sur le réseau de Transputers, pendant qu'un serveur, appelé Cubix, s'exécute sur le processeur Hôte.

Les caractéristiques de cette approche sont:

- Les fonctions d'E/S sont accessibles à tous les programmes résidant sur le réseau de Transputers.

- L'allocation des programmes aux Transputers se fait à partir du serveur Cubix.

L'accès aux unités d'E/S peut être synchrone ou asynchrone.

Dans le mode synchrone, tous les programmes accèdent à la même unité "en même temps". Ce mode se divise en deux sous-modes: le mode simple, dans lequel tous les programmes utilisent les mêmes arguments aux fonctions d'E/S, et le mode multiple, dans lequel les programmes utilisent des arguments différents.

Dans le mode asynchrone, un programme a accès à une unité d'E/S de façon indépendante des autres.

2.3.2.3 Système de communication

Dans le système de communication d'Express, chaque processeur est identifié par un numéro. Le processeur Hôte possède un numéro spécial, alors que le processeur Racine porte le numéro 0. Les numéros des autres processeurs sont déclarés dans un fichier de configuration de la topologie du réseau de Transputers.

Un message circulant entre deux processeurs utilisera un des plus courts chemins entre ces processeurs. Les plus courts chemins sont déterminés de manière statique, avant l'exécution du programme.

Express permet d'utiliser trois grands types de fonctions de communication:

- *Communication "lâchement synchronisée".*
  Lorsqu'un processeur envoit un message, il n'a pas à attendre la réception pour poursuivre l'exécution du programme. On dit que la communication est "lâchement synchronisée", lorsqu'à une fonction de réception doit correspondre une fonction d'envoi ayant des arguments compatibles. Le processeur récepteur suspend ses activités tant qu'il n'a pas reçu le message lui étant destiné.

- *Communication "asynchrone".*
  Les procédures concurrentes sous Express sont implantées comme des fonctions de réception asynchrones. Le processeur attend la réception d'un message particulier pour démarrer une procédure qui s'exécutera concurremment à d'autres. L'utilisation des sémaphores est alors requise pour prévenir les conflits d'accès à la mémoire.

- *Communication par lien physique direct.*
  Il est possible de désactiver temporairement le système de communication d'Express et d'utiliser uniquement les communications par liens physiques directs. On retourne alors au modèle de communication synchronisée décrit à la section 2.1 et également implanté dans Parallel Fortran.

CHAPITRE 3

LE PROBLÈME DE LOCALISATION AVEC ÉQUILIBRAGE

Dans ce chapitre, nous décrivons le problème de localisation avec équilibrage en le situant dans un contexte de planification globale du transport terrestre de conteneurs. Nous présentons ensuite un modèle mathématique pour la résolution du problème, et nous terminons en rappelant les principales approches de résolution proposées dans la littérature.

3.1 Description du problème

Crainic, Dejax et Delorme (1989) exposent ainsi le problème de localisation avec équilibrage:

> Une compagnie de transport de conteneurs livre à ses clients des conteneurs pleins de divers types, par des modes (rail, route, mode combiné, et autres) dont les caractéristiques (coût, temps etc...) sont connues. Pour chaque client, la compagnie connaît le nombre de conteneurs pleins de chaque type à livrer. Lorsque reçus, les clients déchargent les conteneurs, qui sont retournés à un dépôt. Les clients commandent aussi, pour leurs expéditions, des conteneurs vides aux dépôts. Ces demandes sont également connues de l'entreprise. Une fois chargés, les conteneurs sont dirigés vers leur destination, normalement un autre point du réseau de la compagnie, directement ou via leur dépôt d'origine ou un autre. Une complication majeure vient du fait que les demandes des clients entraînent un déséquilibre dans les disponibilités de conteneurs vides entre les divers points du réseau de la compagnie: certaines zones manquent de certains types de conteneurs,

tandis que des surplus sont enregistrés ailleurs. Cette situation impose des transferts de conteneurs vides entre les dépôts afin de rééquilibrer le réseau. Le problème est donc de localiser, parmi un ensemble de sites candidats, les dépôts de conteneurs vides de façon à satisfaire les demandes des clients et à minimiser les coûts totaux d'ouverture et d'opération de tels dépôts ainsi que les coûts de transport clients-dépôts, dépôts-clients et interdépôts.

Ce problème s'inscrit dans la première phase d'une approche biniveau de planification globale du transport terrestre de conteneurs (Crainic, Dejax et Gendreau, 1990). Cette phase ne s'intéresse qu'aux questions d'ordre stratégique et tactique, touchant la planification des mouvements de conteneurs vides: localisation de dépôts, affectation des clients aux dépôts et détermination des mouvements interdépôts. Les décisions prises sont généralement valables pour une période de six mois à un an et sont, pour cette période, constamment utilisées dans la gestion opérationnelle des activités quotidiennes de l'entreprise. Ce dernier niveau de gestion s'intéresse à la fois aux mouvements des conteneurs pleins et vides et constitue la deuxième phase de l'approche de planification globale.

Pour illustrer l'importance des mouvements de conteneurs vides, rappelons le cas d'une compagnie européenne de transport de conteneurs (Crainic, Dejax et Gendreau, 1990). En 1986, on évalua qu'il y eut 250 000 déplacements de conteneurs représentant un coût total estimé de 40 millions de dollars américains. De tous ces déplacements, 40% (95 000) étaient constitués de mouvements de conteneurs vides. Il apparaît donc important, pour une telle compagnie, de réduire les coûts des mouvements de conteneurs vides, qui n'engendrent aucun revenu, de même que le nombre de dépôts nécessaire à l'entreposage des conteneurs vides, ceci dans l'espoir de diminuer les coûts totaux d'ouverture et d'opération de tels dépôts.

Notons que la possibilité de transferts de conteneurs vides entre les dépôts, de même que le caractère bidirectionnel du trafic clients-dépôts, confèrent au problème de localisation avec équilibrage son originalité par rapport à d'autres problèmes de localisation/distribution. Mentionnons en particulier le problème classique de localisation simple, où les mouvements sont unidirectionnels des clients vers les dépôts (Krarup et Pruzan, 1983; Aikens, 1985; Nemhauser et Wolsey, 1988; Cornuejols, Nemhauser et Wolsey, 1990).

3.2 Modélisation

La description du problème, donnée à la section précédente, suggère certaines hypothèses sur le modèle à adopter. Ainsi, les conteneurs vides transitent obligatoirement par un dépôt et il n'y a pas de mouvement direct d'un client vers un autre. De même, les offres et les demandes de chaque client pour chaque type de conteneur sont supposées connues; c'est la composante déterministe du modèle. Il est également raisonnable de supposer que les coûts de transport, ainsi que ceux d'ouverture et d'opération des dépôts, sont linéaires et non-négatifs. Nous supposons enfin que les capacités sur les dépôts, de même que sur les arcs du réseau, ne restreignent pas leur utilisation. Nous considérons donc que les dépôts et les arcs du réseau sont sans capacité.

Sous les hypothèses introduites, il n'est pas nécessaire de considérer explicitement le mode de transport dans la modélisation du problème (Crainic, Dejax et Delorme, 1989). En effet, puisque les coûts sont linéaires et qu'il n'y a pas de capacité sur les arcs, lorsqu'un conteneur d'un type donné peut être transporté entre deux points du réseau au moyen de plusieurs modes de transport, il utilisera celui de moindre coût unitaire. Nous négligeons donc la composante mode de transport dans la modélisation du problème.

Dans la suite du texte, nous désignerons un type de conteneur donné par le terme "produit".

Définissons les ensembles suivants, ainsi que leur cardinalité:

$C = \{i \mid \text{client}\}$;

$n = |C|$;

$D = \{j \mid \text{dépôt}\}$;

$m = |D|$;

$P = \{p \mid \text{produit}\}$;

$l = |P|$.

Définissons également les données et les variables de décision:

O_{ip}	:	offre du client i pour le produit p, $i \in C, p \in P$;
D_{ip}	:	demande du client i pour le produit p, $i \in C, p \in P$;
f_j	:	coût d'ouverture et d'opération du dépôt j, $j \in D$;
$c_{ijp}(c_{jip})$	:	coût unitaire de transport du client i au dépôt j (du dépôt j au client i) pour le produit p, $i \in C, j \in D, p \in P$;
s_{jkp}	:	coût unitaire de transport du dépôt j au dépôt k pour le produit p, $j,k \in D, p \in P$;
$y = (y_j)$	:	vecteur d'ouverture des dépôts, $j \in D$; $y_j = \begin{cases} 1, & \text{si le dépôt } j \text{ est ouvert} \\ 0, & \text{sinon} \end{cases}$
$x = (x_{ijp}, x_{jip})$	:	vecteur des flots circulant entre le client i et le dépôt j pour le produit p, $i \in C, j \in D, p \in P$;
$w = (w_{jkp})$	:	vecteur des flots circulant entre les dépôts j et k pour le produit p, $j,k \in D, p \in P$.

Comme l'illustre la figure 14, les éléments du modèle ainsi définis permettent d'entrevoir la structure de réseau présente dans le problème. Conceptuellement, nous sommes en présence d'un réseau $G = (N, A)$, où $N = C \cup D$ et $A \subseteq (C \times D \times P) \cup (D \times D \times P) \cup (D \times C \times P)$.

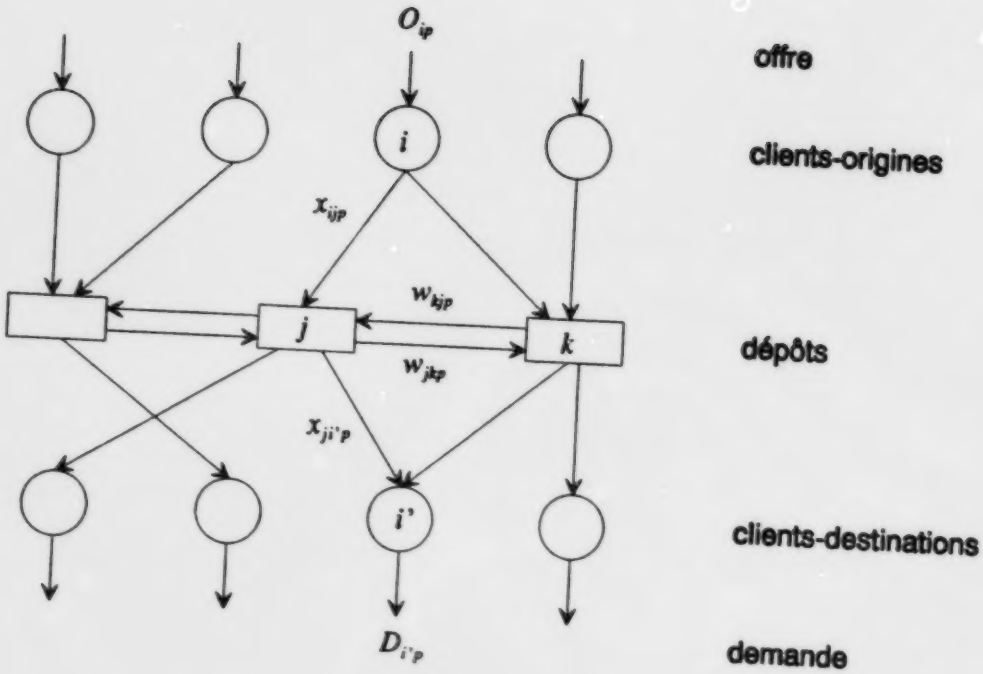

Figure 14. Structure de réseau du problème de localisation avec équilibrage

Nous pouvons maintenant introduire un modèle linéaire, déterministe et sans capacité pour le problème de localisation avec équilibrage (Crainic, Dejax et Delorme, 1989):

$$(Q) \quad \textit{Minimiser} \quad z = \sum_{j \in D} f_j y_j + \sum_{p \in P} \sum_{j \in D} \left(\sum_{i \in C} (c_{ijp} x_{ijp} + c_{jip} x_{jip}) + \sum_{k \in D} s_{jkp} w_{jkp} \right)$$

sujet à

$$\sum_{j \in D} x_{ijp} = O_{ip} \qquad \forall i \in C, p \in P$$

$$\sum_{j \in D} x_{jip} = D_{ip} \qquad \forall i \in C, p \in P \qquad (Q1)$$

$$x_{ijp} \leq M y_j \qquad \forall i \in C, j \in D, p \in P$$

$$x_{jip} \leq M y_j \qquad \forall i \in C, j \in D, p \in P \qquad (Q2)$$

$$w_{jkp} \leq M y_j \qquad \forall j, k \in D, p \in P$$

$$w_{kjp} \leq M y_j \qquad \forall j, k \in D, p \in P \qquad (Q3)$$

$$\sum_{i \in C} x_{ijp} + \sum_{k \in D} w_{kjp} - \sum_{i \in C} x_{jip} - \sum_{k \in D} w_{jkp} = 0 \qquad \forall j \in D, p \in P \qquad (Q4)$$

$$x_{ijp} \geq 0 \qquad \forall i \in C, j \in D, p \in P$$

$$x_{jip} \geq 0 \qquad \forall i \in C, j \in D, p \in P$$

$$w_{jkp} \geq 0 \qquad \forall j, k \in D, p \in P$$

$$y_j \in \{0, 1\} \qquad \forall j \in D$$

Les contraintes (Q1) assurent que les offres et les demandes des clients sont satisfaites. Les contraintes (Q2) interdisent les mouvements entre les clients et les dépôts fermés (M est une constante positive, suffisamment grande), alors que les contraintes (Q3) jouent un rôle similaire pour les mouvements interdépôts. Les contraintes (Q4) assurent l'équilibrage des flots transitant par les dépôts.

Utilisant une hypothèse réaliste sur les coûts unitaires de transport entre les dépôts, dite hypothèse d'inégalité du triangle, Crainic, Dejax et Delorme (1989) démontrent la proposition suivante:

<u>Proposition 3.1</u>

Si

$$s_{jkp} \leq s_{jrp} + s_{rkp} \qquad ,\forall j,k,r \in D, p \in P,$$

alors les contraintes (Q3) du modèle (Q) sont redondantes.

En supposant l'hypothèse d'inégalité du triangle vérifiée en pratique, et en remplaçant dans chaque contrainte de type (Q2) la constante M par O_{ip} et D_{ip}, de façon à restreindre le domaine, nous obtenons le modèle simplifié suivant (Crainic, Dejax et Delorme, 1989):

(L) *Minimiser*
$$z = \sum_{j \in D} f_j y_j + \sum_{p \in P} \sum_{j \in D} \left(\sum_{i \in C} (c_{ijp} x_{ijp} + c_{jip} x_{jip}) + \sum_{k \in D} s_{jkp} w_{jkp} \right)$$

sujet à

$$\sum_{j \in D} x_{ijp} = O_{ip} \qquad \forall i \in C, p \in P$$

$$\sum_{j \in D} x_{jip} = D_{ip} \qquad \forall i \in C, p \in P \qquad \text{(L1)}$$

$$x_{ijp} \leq O_{ip} y_j \qquad \forall i \in C, j \in D, p \in P$$

$$x_{jip} \leq D_{ip} y_j \qquad \forall i \in C, j \in D, p \in P \qquad \text{(L2)}$$

$$\sum_{i \in C} x_{ijp} + \sum_{k \in D} w_{kjp} - \sum_{i \in C} x_{jip} - \sum_{k \in D} w_{jkp} = 0 \qquad \forall j \in D, p \in P \qquad \text{(L3)}$$

$$x_{ijp} \geq 0 \qquad \forall i \in C, j \in D, p \in P$$

$$x_{jip} \geq 0 \qquad \forall i \in C, j \in D, p \in P$$

$$w_{jkp} \geq 0 \qquad \forall j,k \in D, p \in P$$

$$y_j \in \{0,1\} \qquad \forall j \in D$$

3.3 Approches de résolution

La présence des variables entières de localisation rend difficile la résolution du problème. C'est pourquoi les propriétés de sous-problèmes issus du modèle (L) ont été étudiées (Crainic, Dejax et Delorme, 1989). Dans tous ces sous-problèmes, émerge une structure de réseau favorable aux développements algorithmiques. Ceux-ci utilisent plusieurs méthodes de programmation mathématique: un algorithme de séparation et évaluation progressive (Hodgson, 1988; Crainic, Delorme et Dejax, 1989), une technique de résolution basée sur le dual du modèle (Delorme, Crainic et Dejax, 1988), une approche de restriction basée sur une méthode primale-duale (Tétreault, 1990), des méthodes de relaxation lagrangienne, ainsi que des heuristiques (Guo, 1990). Plusieurs de ces méthodes sont encore à l'essai et déjà, d'autres développements algorithmiques pointent à l'horizon: combinaison de la structure de séparation et évaluation progressive et de l'algorithme basé sur le dual du modèle, utilisation de la méthode de recherche Tabu (Glover, 1986), etc...

Dans ce travail, nous ne nous intéressons qu'à l'algorithme de séparation et évaluation progressive, dont une description générale fait l'objet du prochain chapitre.

CHAPITRE 4

ALGORITHME DE SÉPARATION ET ÉVALUATION PROGRESSIVE

Dans ce chapitre, nous nous intéressons à un algorithme de résolution pour le problème de localisation avec équilibrage, algorithme basé sur la méthode générale de séparation et évaluation progressive. Nous décrivons d'abord la forme générale de cette méthode, appliquée à un problème de programmation mixte. Nous étudions ensuite l'adaptation qui en a été faite pour résoudre le problème de localisation avec équilibrage.

4.1 Algorithme général de séparation et évaluation progressive

Considérons le problème de minimisation suivant:

$$(P) \qquad \textit{Minimiser} \quad f(x)$$
$$\textit{sujet à} \quad x \in S,$$

où $S \subseteq X$ désigne le domaine réalisable et X l'espace des variables. Nous nous restreignons ici à l'étude de problèmes de la forme (P) pour lesquels certaines composantes de $x \in S$ sont contraintes à prendre leurs valeurs dans un ensemble discret fini ou infini dénombrable. Nous supposons également que f est une fonction à valeurs réelles.

La résolution d'un tel problème est compliquée du fait de la nature discrète imposée à certaines variables. Pour remédier à cette difficulté, la méthode générale de séparation et évaluation progressive consiste en premier lieu, à séparer l'espace $X = X_0$ en k sous-ensembles $X_1^0, \ldots, X_k^0$ mutuellement disjoints, ce qui induit une partition du domaine réalisable $S = S_0$ en k sous-ensembles $S_1^0, \ldots, S_k^0$. Ce processus génère k sous-problèmes de la forme:

(P_i^0) *Minimiser* $f(x)$

sujet à $x \in S_i^0$

À leur tour, ces sous-problèmes peuvent être d'une complexité aussi grande que le problème original. Les ensembles X_i^0 peuvent alors être séparés en l_i sous-ensembles $X_1^i, \ldots, X_{l_i}^i$, procédé qui génère l_i nouveaux sous-problèmes à résoudre.

Dans la suite du texte, notons P_i un sous-problème généré à une étape du processus et X_i et S_i, les sous-ensembles correspondants.

L'objectif du processus de séparation est d'obtenir des sous-problèmes pour lesquels une solution optimale est connue ou dont on peut déterminer que leur domaine réalisable est vide. Dans ces deux cas, il est inutile de séparer ces sous-problèmes. Dans le premier cas, la solution optimale déterminée est réalisable pour le problème original $P = P_0$ et l'objectif f évalué en cette solution nous fournit alors une **borne supérieure** sur la valeur optimale de $P = P_0$. Cette constatation nous permet d'identifier un troisième critère permettant l'arrêt du processus de séparation pour un sous-problème donné: si une **borne inférieure** sur la valeur optimale du sous-problème est connue et que cette borne dépasse la meilleure borne supérieure z identifiée, il est alors inutile de poursuivre le processus de séparation. Ceci est dû au fait que pour un sous-problème donné, sa valeur optimale est bornée supérieurement par la valeur optimale de chacun de ses sous-problèmes.

Pour obtenir une borne inférieure sur la valeur optimale du sous-problème P_i, une relaxation $\overline{P_i}$ sera résolue:

$(\overline{P_i})$ *Minimiser* $g(x)$

sujet à $x \in \overline{S_i}$,

où $S_i \subseteq \overline{S_i} \subseteq X_i$ et $g(x) \leq f(x), \forall x \in S_i$.

En plus d'être plus facile à résoudre que le problème P_i, cette relaxation doit posséder la propriété que lorsqu'une de ses solutions optimales résulte en une solution

réalisable pour le problème P_i (et donc pour le problème original $P = P_0$), cette solution est également optimale pour le problème P_i. Évidemment, dans le cas où le domaine réalisable de la relaxation $\overline{P_i}$ est vide, celui du problème P_i l'est également.

Le processus de séparation, de même que les trois critères d'arrêt de séparation, permettent de générer un **arbre de recherche** d'une solution optimale, appelé arbre de séparation et évaluation progressive. À la racine de l'arbre, se trouve le problème original. Les problèmes générés mais non-examinés sont ceux dont les fils n'ont pas été générés ou dont on n'a pas déterminé s'ils satisfont un des critères d'arrêt de séparation.

La figure 15 illustre un arbre de séparation et évaluation progressive. Les problèmes générés et examinés sont représentés par les noeuds 1 et 2. Le problème 4 est le seul qui satisfait un des critères d'arrêt de séparation; il est représenté par une ligne double sous le cercle. Les problèmes 3, 5 et 6 sont générés mais non-examinés; leurs fils n'ont pas encore été générés.

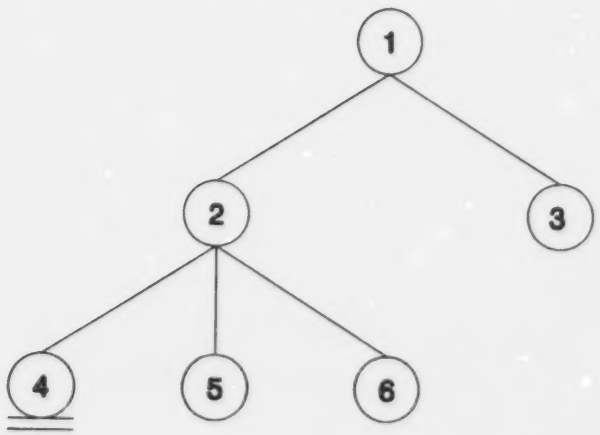

Figure 15. Exemple d'arbre de séparation et évaluation progressive

Nous pouvons maintenant donner l'algorithme de séparation et évaluation progressive pour trouver une solution optimale au problème P. Notons Γ l'ensemble des problèmes générés mais non-examinés et z la meilleure borne supérieure connue sur la valeur optimale de P.

Algorithme 4.1: Algorithme de séparation et évaluation progressive pour trouver une solution optimale au problème P.

Étape 1 : $\Gamma := \{P\}$; $z := \infty$.

Étape 2 : Choisir un problème $P_i \in \Gamma$ et $\Gamma := \Gamma - \{P_i\}$. Résoudre sa relaxation $\overline{P_i}$.

Étape 3 : Si $\overline{S_i} = \emptyset$, alors aller à l'étape 7.

Étape 4 : Soient $\overline{x}$, solution optimale de $\overline{P_i}$ et $g(\overline{x})$, sa valeur optimale.
Si $\overline{x} \in S_i$, alors :
si $g(\overline{x}) < z$, alors $z := g(\overline{x})$;
aller à l'étape 7.

Étape 5 : Si $g(\overline{x}) \geq z$, alors aller à l'étape 7.

Étape 6 : Partitionner le problème P_i en l_i sous-problèmes $P_1^i, \ldots, P_{l_i}^i$ et $\Gamma := \Gamma \cup \{P_1^i, \ldots, P_{l_i}^i\}$. Aller à l'étape 2.

Étape 7 : Si $\Gamma \neq \emptyset$, alors aller à l'étape 2; sinon, l'algorithme se termine et:
si $z < \infty$, alors la solution x^* satisfaisant $z = f(x^*)$ est optimale;
si $z = \infty$, alors le domaine réalisable est vide.

Il convient de mentionner une variante importante de cet algorithme. À l'étape 2, il est possible de résoudre la relaxation de plusieurs problèmes à la fois, et de choisir parmi ceux-ci, celui dont la relaxation fournit la meilleure borne. Cette règle pour choisir un problème à examiner est communément appelée "règle de la meilleure borne".

Le choix d'un problème à l'étape 2, de même que le partitionnement à l'étape 6 et la qualité de la relaxation, sont des paramètres cruciaux pour mesurer l'efficacité de l'algorithme. Une discussion détaillée de ces sujets se trouve dans Ibaraki (1987).

4.2 Algorithme de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage

Se rapportant au modèle (L) (voir section 3.2), nous remarquons que les seules variables discrètes sont les y_j, qui sont restreintes à prendre les valeurs 0 ou 1. Il s'ensuit qu'une partition naturelle de l'espace des variables consiste à choisir un y_{j^*} et à considérer les sous-problèmes où $y_{j^*} = 0$ et $y_{j^*} = 1$. Cette partition induit un arbre binaire de recherche.

Le calcul d'une borne inférieure à chaque noeud de l'arbre se fait au moyen d'une relaxation, appelée **relaxation faible**, dont nous décrivons les propriétés à la prochaine sous-section. Nous montrons également qu'il est possible d'identifier une borne supérieure sans que la solution de la relaxation faible soit réalisable. La stratégie de parcours de l'arbre (étape 2 de l'algorithme 4.1) est ensuite étudiée, puis nous constatons que le processus de partitionnement (étape 6 de l'algorithme 4.1) consiste à choisir une variable y_{j^*} et à générer deux nouveaux sous-problèmes. Le choix de la variable y_{j^*} se fait au moyen de règles de branchement. Nous en étudions quelques-unes. Nous résumons ensuite la section en donnant une version de l'algorithme et nous terminons en traitant des détails de l'implantation réalisée.

4.2.1 Relaxation faible

La relaxation faible du modèle (L) (voir section 3.2) consiste à abandonner les contraintes d'intégralité sur les y_j et à agréger les contraintes complicantes de type (L2) (Crainic, Dejax et Delorme, 1989):

(R) *Minimiser* $$z = \sum_{j \in D} f_j y_j + \sum_{p \in P} \sum_{j \in D} \left(\sum_{i \in C} (c_{ijp} x_{ijp} + c_{jip} x_{jip}) + \sum_{k \in D} s_{jkp} w_{jkp} \right)$$

sujet à

$$\sum_{j \in D} x_{ijp} = O_{ip} \qquad \forall i \in C, p \in P$$

$$\sum_{j \in D} x_{jip} = D_{ip} \qquad \forall i \in C, p \in P \qquad \text{(R1)}$$

$$\sum_{p \in P} \sum_{i \in C} (x_{ijp} + x_{jip}) \leq \sum_{p \in P} \sum_{i \in C} (O_{ip} + D_{ip}) y_j \qquad \forall j \in D \qquad \text{(R2)}$$

$$\sum_{i \in C} x_{ijp} + \sum_{k \in D} w_{jkp} - \sum_{i \in C} x_{jip} - \sum_{k \in D} w_{jkp} = 0 \qquad \forall j \in D, p \in P \qquad \text{(R3)}$$

$$x_{ijp} \geq 0 \qquad \forall i \in C, j \in D, p \in P$$
$$x_{jip} \geq 0 \qquad \forall i \in C, j \in D, p \in P$$
$$w_{jkp} \geq 0 \qquad \forall j, k \in D, p \in P$$
$$y_j \geq 0 \qquad \forall j \in D$$

Une formulation de relaxation faible est utilisée pour déterminer une borne inférieure à chaque noeud de l'arbre de recherche. Cette formulation doit tenir compte du partitionnement en sous-problèmes, caractérisé par l'état des variables y_j au noeud courant, état défini par les trois ensembles suivants:

$$K_0 = \{ j \mid y_j = 0, j \in D \},$$
$$K_1 = \{ j \mid y_j = 1, j \in D \},$$
$$K_{01} = \{ j \mid j \notin K_0 \cup K_1, j \in D \}.$$

K_0 désigne l'ensemble des dépôts dits fermés, K_1 l'ensemble des dépôts dits ouverts et K_{01} l'ensemble des dépôts dits libres.

La relaxation faible, à un noeud donné de l'arbre de séparation et évaluation progressive, peut donc être formulée ainsi (Crainic, Delorme et Dejax, 1989):

(RF) *Minimiser* $z = \sum_{j \in K_1} f_j + \sum_{j \in K_{01}} f_j y_j +$

$$\sum_{p \in P} \sum_{j \in K_1 \cup K_{01}} \left(\sum_{i \in C} (c_{ijp} x_{ijp} + c_{jip} x_{jip}) + \sum_{k \in K_1 \cup K_{01}} s_{jkp} w_{jkp} \right)$$

sujet à

$$\sum_{j \in K_1 \cup K_{01}} x_{ijp} = O_{ip} \qquad \forall i \in C, p \in P$$

$$\sum_{j \in K_1 \cup K_{01}} x_{jip} = D_{ip} \qquad \forall i \in C, p \in P \qquad \text{(RF1)}$$

$$\sum_{p \in P} \sum_{i \in C} (x_{ijp} + x_{jip}) \leq \sum_{p \in P} \sum_{i \in C} (O_{ip} + D_{ip}) \qquad \forall j \in K_1 \qquad \text{(RF2)}$$

$$\sum_{p \in P} \sum_{i \in C} (x_{ijp} + x_{jip}) \leq \sum_{p \in P} \sum_{i \in C} (O_{ip} + D_{ip}) y_j \qquad \forall j \in K_{01} \qquad \text{(RF3)}$$

$$\sum_{i \in C} x_{ijp} + \sum_{k \in D} w_{kjp} - \sum_{i \in C} x_{jip} - \sum_{k \in D} w_{jkp} = 0 \qquad \forall j \in K_1 \cup K_{01}, p \in P \qquad \text{(RF4)}$$

$$x_{ijp} = 0 \qquad \forall i \in C, j \in K_0, p \in P$$

$$x_{jip} = 0 \qquad \forall i \in C, j \in K_0, p \in P$$

$$x_{ijp} \geq 0 \qquad \forall i \in C, j \in K_1 \cup K_{01}, p \in P$$

$$x_{jip} \geq 0 \qquad \forall i \in C, j \in K_1 \cup K_{01}, p \in P$$

$$w_{jkp} = 0 \qquad \forall j \in D, k \in K_0, p \in P \quad \text{ou}$$

$$\forall j \in K_0, k \in D, p \in P$$

$$w_{jkp} \geq 0 \qquad \forall j, k \in K_1 \cup K_{01}, p \in P$$

$$y_j \geq 0 \qquad \forall j \in K_{01}$$

Dans la suite, notons $(\overline{x}, \overline{w}, \overline{y})$ une solution optimale de (RF) et définissons l'ensemble suivant: $\overline{K_{01}} = \{j \mid \overline{y_j} > 0, j \in K_{01}\}$.

La proposition suivante, dont une preuve se trouve dans Hodgson (1988), permet de simplifier le modèle (RF):

Proposition 4.1

Une solution optimale du problème (RF) satisfait les contraintes (RF3) avec égalité.

Cette proposition implique que:

$$\overline{y_j} = \frac{\sum_{p \in P} \sum_{i \in C} (\overline{x_{ijp}} + \overline{x_{jip}})}{\sum_{p \in P} \sum_{i \in C} (O_{ip} + D_{ip})} \quad , \forall j \in K_{01}.$$

Utilisant cette simplification, de même que le fait que les contraintes (RF2) sont redondantes à cause de (RF1) et des contraintes de non-négativité, le modèle peut se réécrire (Crainic, Delorme et Dejax, 1989):

(PR) *Minimiser*

$$z = \sum_{j \in K_1} f_j + \sum_{p \in P} \Big(\sum_{i \in C} \sum_{j \in K_1} (c_{ijp} x_{ijp} + c_{jip} x_{jip}) + \sum_{i \in C} \sum_{j \in K_{01}} \left\{ \left(c_{ijp} + \frac{f_j}{\Delta} \right) x_{ijp} + \left(c_{jip} + \frac{f_j}{\Delta} \right) x_{jip} \right\} + \sum_{j \in K_1 \cup K_{01}} \sum_{k \in K_1 \cup K_{01}} s_{jkp} w_{jkp} \Big)$$

sujet à

$$\sum_{j \in K_1 \cup K_{01}} x_{ijp} = O_{ip} \qquad \forall i \in C, p \in P$$

$$\sum_{j \in K_1 \cup K_{01}} x_{jip} = D_{ip} \qquad \forall i \in C, p \in P \qquad \text{(PR1)}$$

$$\sum_{i \in C} x_{ijp} + \sum_{k \in D} w_{kjp} - \sum_{i \in C} x_{jip} - \sum_{k \in D} w_{jkp} = 0 \qquad \forall j \in K_1 \cup K_{01}, p \in P \qquad \text{(PR2)}$$

$$x_{ijp} = 0 \qquad \forall i \in C, j \in K_0, p \in P$$

$$x_{jip} = 0 \qquad \forall i \in C, j \in K_0, p \in P$$

$$x_{ijp} \geq 0 \qquad \forall i \in C, j \in K_1 \cup K_{01}, p \in P$$

$$x_{jip} \geq 0 \qquad \forall i \in C, j \in K_1 \cup K_{01}, p \in P$$

$$w_{jkp} = 0 \qquad \forall j \in D, k \in K_0, p \in P \quad \text{ou} \quad \forall j \in K_0, k \in D, p \in P$$

$$w_{jkp} \geq 0 \qquad \forall j, k \in K_1 \cup K_{01}, p \in P$$

où $\Delta = \sum_{p \in P} \sum_{i \in C} (O_{ip} + D_{ip})$.

La proposition suivante, dont une preuve se trouve dans Crainic, Dejax et Delorme (1989), justifie l'utilisation de la relaxation faible pour calculer une borne inférieure à chaque noeud de l'arbre de séparation et évaluation progressive:

Proposition 4.2
(PR) est un problème de multiflot à coût minimum.

Notons que (PR) est décomposable en l problèmes de flot à coût minimum mutuellement indépendants. Cette propriété est intéressante car il existe plusieurs techniques connues de résolution d'un problème de flot à coût minimum.

La proposition suivante, dont une preuve se trouve dans Hodgson (1988), nous renseigne sur la qualité de la borne inférieure calculée par la relaxation faible:

Proposition 4.3
À tout noeud de l'arbre de séparation et évaluation progressive,

$$\sum_{j \in K_{01}} \overline{y_j} \leq 1.$$

Cette proposition explique en partie pourquoi la relaxation faible peut fournir une borne inférieure relativement faible: elle diminue l'apport des coûts fixes d'ouverture et d'opération des dépôts dans la fonction économique. Cette proposition limite aussi le domaine de définition des variables y_j puisqu'elle implique que $0 \leq \overline{y_j} \leq 1$.

Finalement, la proposition suivante (Hodgson, 1988; Crainic, Delorme et Dejax, 1989) nous assure que la relaxation faible possède bien la propriété que lorsque sa solution optimale résulte en une solution réalisable pour le sous-problème considéré, elle est également optimale pour ce sous-problème:

Proposition 4.4
Si la solution optimale de (PR), correspondant à un noeud donné de l'arbre de séparation et évaluation progressive, résulte en des valeurs entières des y_j, alors:
i) cette solution est réalisable pour le problème original (L);
ii) cette solution est optimale pour le sous-problème considéré à ce noeud de l'arbre.

4.2.2 Borne supérieure

La proposition précédente nous indique que lorsque la solution optimale de la relaxation faible est entière, elle est réalisable pour le problème original et sa valeur optimale fournit alors une borne supérieure. Cependant, il est possible de calculer une borne supérieure même si la solution n'est pas entière: il suffit de poser

$$y_j^{\cdot} = \begin{cases} 1, & \text{si } \overline{y_j} > 0 \\ 0, & \text{sinon} \end{cases}$$

Si cette borne supérieure calculée est inférieure à la meilleure borne supérieure connue, elle la remplace.

4.2.3 Stratégie de parcours de l'arbre

Le choix d'une stratégie de parcours de l'arbre de séparation et évaluation progressive doit être guidé par des critères d'économie du temps de calcul et d'efficacité dans la recherche d'une solution.

Dans notre cas, le calcul d'une borne inférieure nécessite la résolution d'un problème de multiflot à coût minimum, qui requiert, pour des applications réelles, un temps d'exécution non-négligeable. De plus, à cause de la faiblesse de la borne inférieure ainsi calculée (voir proposition 4.3), il n'est généralement pas réaliste, pour des problèmes réels, d'explorer complètement l'arbre de séparation et évaluation progressive. On se contentera plutôt d'explorer l'arbre jusqu'à l'obtention d'une "bonne" solution.

Ainsi, le choix d'un sous-problème à examiner selon la règle de la meilleure borne (voir section 4.1) n'est pas économique. En effet, cette règle implique le calcul de la borne inférieure pour les deux sous-problèmes générés à chaque itération de même qu'un ordonnancement des sous-problèmes non-examinés. L'avantage principal de cette règle est qu'elle permet de minimiser le nombre de sous-problèmes examinés dans l'arbre complet de séparation et évaluation progressive (Ibaraki, 1987). Cet avantage s'efface totalement dans notre cas, puisque l'exploration complète de l'arbre se fera rarement. Pire encore, en utilisant cette règle, une solution optimale n'est généralement trouvée que dans les derniers noeuds explorés (Ibaraki, 1987).

De même, une stratégie de recherche en largeur, parcourant l'arbre par niveau, ne serait ni économique ni efficace, à cause de l'accroissement exponentiel du nombre de noeuds par niveau.

Par contre, une stratégie de recherche en profondeur possède de nombreux avantages. Cette stratégie sélectionne parmi les noeuds générés mais non-examinés, un de ceux qui se trouvent à plus grande distance de la racine. En plus d'être facile à implanter au moyen d'une pile LIFO ("last in, first out"), elle permet d'identifier rapidement des solutions réalisables car celles-ci sont plus susceptibles de se trouver profondément dans l'arbre que près de la racine (Nemhauser et Wolsey, 1988). Ceci permet d'obtenir rapidement de meilleures bornes supérieures et, en cas d'arrêt de l'algorithme avant l'énumération complète, on peut espérer avoir atteint une "bonne" solution réalisable.

La figure 16 schématise un arbre binaire créé par un algorithme de séparation et évaluation progressive utilisant une stratégie de recherche en profondeur. Les noeuds de l'arbre sont numérotés dans l'ordre où ils sont évalués.

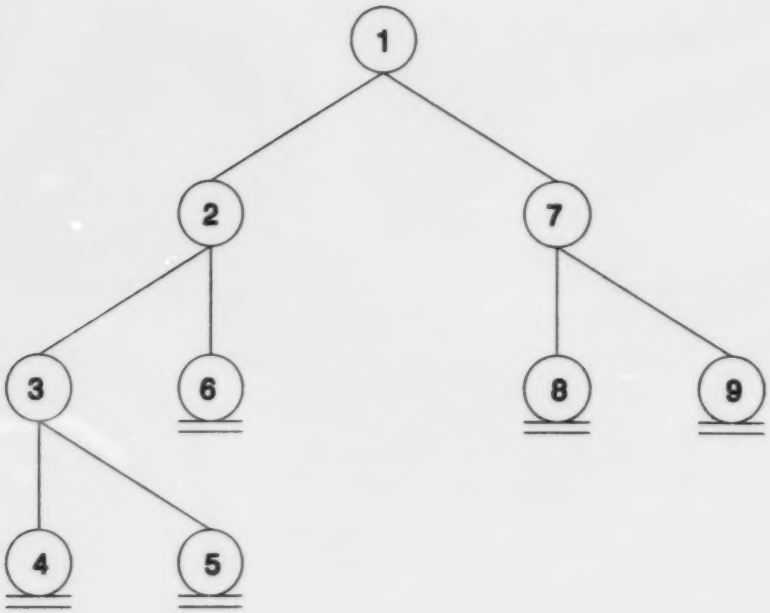

Figure 16. Exemple d'arbre binaire de séparation et évaluation progressive créé par un parcours en profondeur

4.2.4 Règles de branchement

La génération de sous-problèmes et le choix de celui à examiner prioritairement, se font au moyen d'un critère de sélection d'une variable à fixer, appelé aussi règle de branchement. Cette règle permet de sélectionner un $y_{j^*}, j^* \in \overline{K_{01}}$, et d'examiner prioritairement soit le sous-problème où $y_{j^*} = 0$, soit celui où $y_{j^*} = 1$. Dans le premier cas, nous parlerons d'une règle de branchement à 0, et dans le second, d'une règle de branchement à 1.

L'importance des règles de branchement a été mise en lumière par les travaux de Khumawala (1972). Celui-ci a mis au point une implantation de l'algorithme de séparation et évaluation progressive proposé par Effroymson et Ray (1966) pour résoudre le problème de localisation simple. Cette implantation a permis de tester plusieurs règles de branchement et d'établir un classement entre elles selon leur efficacité dans l'exploration de l'arbre de séparation et évaluation progressive.

Ce classement a été utilisé pour traiter des généralisations du problème de localisation simple (Kaufman, Eede et Hansen, 1977, Neebe et Khumawala, 1981; Ro et Tcha, 1984).

Crainic, Delorme et Dejax (1989) ont conçu des règles de branchement adaptées à la résolution du problème de localisation avec équilibrage. Certaines de ces règles sont basées sur les travaux de Khumawala. En tout, 13 règles originelles ont été identifiées, puis une quatorzième s'est ajoutée lors des essais numériques. De ceux-ci, découlent les conclusions suivantes:

- Les adaptations des règles identifiées comme les meilleures par Khumawala pour résoudre le problème de localisation simple, ne sont pas efficaces pour le problème de localisation avec équilibrage.

- Les règles de branchement à 0 sont généralement efficaces en terme de temps de calcul. Ceci est dû au fait que lorsqu'un dépôt est fixé fermé, la taille du réseau est diminuée, puisque les arcs entre ce dépôt et les clients sont éliminés.

- Les règles de branchement à 0 donnent généralement de bons résultats pour des problèmes simples pour lesquels le nombre de produits est peu élevé (un ou deux). Par contre, lorsque la complexité du problème augmente (trois produits ou plus, et davantage de déséquilibre entre les zones), la relaxation faible identifie très peu de solutions réalisables et l'efficacité de ces règles à trouver une "bonne" solution s'en ressent grandement.

- Les règles de branchement à 1 obtiennent généralement de meilleures solutions surtout lorsque la complexité du problème augmente. Ceci est dû au fait qu'un nombre élevé de produits et un déséquilibre accru entre les zones indiquent qu'une grande proportion des dépôts doivent être ouverts. Ces règles permettent également d'identifier rapidement des solutions réalisables.

- Parmi ces règles, les meilleures sont les règles de quotient:

  Déterminer $j^* = \underset{j \in \overline{K_{01}}}{ArgMin} \{[f_j + C_j(\overline{w_j}, \overline{x_j})] / \overline{y_j}\}$ et explorer prioritairement le noeud

  où $y_{j^*} = 1$; $C_j(\overline{w_j}, \overline{x_j})$ mesure un certain coût associé au transport des conteneurs transitant par le dépôt j.

  Selon le cas, on évaluera:

  - $C_j(\overline{w_j}, \overline{x_j}) = 0$: c'est la règle du quotient simplifié (SSQ);

  - $C_j(\overline{w_j}, \overline{x_j}) = \sum_{p \in P} \sum_{k \in D} (s_{jkp}\overline{w_{jkp}} + s_{kjp}\overline{w_{kjp}})$: c'est la règle du quotient modifié (SMQ);

  - $C_j(\overline{w_j}, \overline{x_j}) = \sum_{p \in P} \sum_{k \in D} (s_{jkp}\overline{w_{jkp}} + s_{kjp}\overline{w_{kjp}}) + \sum_{p \in P} \sum_{i \in C} (c_{ijp}\overline{x_{ijp}} + c_{jip}\overline{x_{jip}})$: c'est la règle Z (SZ).

La motivation derrière l'emploi des règles de quotient est qu'elles permettent de comparer les coûts engendrés par l'ouverture d'un dépôt ($C_j(\overline{w_j},\overline{x_j})$) à l'activité "positive" générée par ce dépôt ($\overline{y_j}$), ceci en se basant sur le résultat de la proposition 4.1.

Il est également possible de combiner l'efficacité d'un critère de branchement à 0 avec la qualité d'une règle de quotient, au moyen de règles hybrides. Celles-ci sont obtenues en utilisant une règle de branchement à 0 lorsque la valeur d'une certaine mesure associée à un dépôt libre, dépasse un seuil fixé à l'avance. Lorsque le seuil n'est pas dépassé, une règle de quotient est utilisée. Les seuils doivent être conçus de telle sorte que la règle de branchement à 0 soit privilégiée au début de l'énumération, pour permettre une diminution rapide de la borne supérieure et l'identification des dépôts les plus susceptibles d'être fermés dans la solution optimale.

Deux approches ont été testées:

- Seuil ε : Si $y_{j^*} = \min_{j \in \overline{K_{01}}} \overline{y_j} < \varepsilon$, alors choisir le dépôt j^* et explorer prioritairement le noeud où $y_{j^*} = 0$.

  Une valeur appropriée pour le seuil est $\varepsilon = 1/2m$. Cette valeur représente la moitié de la moyenne des y_j, lorsqu'aucun des y_j n'a encore été fixé, à la racine de l'arbre de séparation et évaluation progressive.

- Seuil M : Si $F_{j^*} = \max_{j \in \overline{K_{01}}} \{f_j/\overline{y_j}\} > M$, alors choisir le dépôt j^* et explorer prioritairement le noeud où $y_{j^*} = 0$.

  Une valeur possible pour M consiste à combiner la moyenne des coûts fixes avec le plus grand coût fixe: $M = m \max_{j \in D} \{f_j\} + \sum_{j \in D} f_j$.

Les seuils ont été combinés avec les règles de quotient pour générer six règles hybrides. Des essais numériques effectués sur ces règles, se dégagent les conclusions suivantes:

- Lorsque le nombre de produits est petit (cinq et moins), les règles hybrides trouvent de meilleures solutions plus rapidement que les règles de quotient.

- Lorsque le nombre de produits augmente, les règles hybrides sont déclassées par les règles de quotient.

- Les règles hybrides utilisant le seuil M sont généralement plus efficaces que celles utilisant le seuil ε.

4.2.5 Algorithme

Nous pouvons maintenant donner une version de l'algorithme de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage.

Notons Λ la pile des problèmes générés mais non-examinés. Une sous-pile est notée entre [ ], le problème le plus à droite entre les [ ] se trouvant alors au-dessus de la pile. Les opérateurs $\oplus$ et $\ominus$ désignent respectivement la suppression et l'ajout de problèmes sur la pile. Notons L_0 le problème original issu du modèle (L) et L_i un sous-problème généré à une étape quelconque. Sa relaxation faible est notée R_i. $D(P)$ désigne le domaine réalisable d'un problème P quelconque. Enfin, notons z_{sup} la meilleure borne supérieure connue et $z(x, w, y)$, la valeur de l'objectif de L_0 au point réalisable (x, w, y).

Algorithme 4.2: Algorithme de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage.

Étape 1 : $\Lambda := [L_0]$; $z_{sup} := \infty$.

Étape 2 : $\Lambda := \Lambda \ominus [L_i]$. Résoudre la relaxation faible R_i.

Étape 3 : Si $D(R_i) = \emptyset$, alors aller à l'étape 8.

Étape 4 : Soient $(\overline{x}, \overline{w}, \overline{y})$, solution optimale de R_i et $\overline{z}$, sa valeur optimale. Si $(\overline{x}, \overline{w}, \overline{y}) \in D(L_i)$, alors:
si $\overline{z} < z_{sup}$, alors $z_{sup} := \overline{z}$;
aller à l'étape 8.

Étape 5 : Si $\overline{z} \geq z_{sup}$, alors aller à l'étape 8.

Étape 6 : $z' := z(\overline{x}, \overline{w}, y')$, où

$$y_j' = \begin{cases} 1, & \text{si } \overline{y_j} > 0 \\ 0, & \text{sinon} \end{cases}$$

Si $z' < z_{sup}$, alors $z_{sup} := z'$.

Étape 7 : Choisir une variable $y_{j^*}, j^* \in \overline{K_{01}}$ et créer les deux sous-problèmes $L^i_{y_{j^*}=0}$, $L^i_{y_{j^*}=1}$. Si la règle de branchement est à 0, faire $\Lambda := \Lambda \oplus \left[L^i_{y_{j^*}=1}, L^i_{y_{j^*}=0}\right]$; sinon, faire $\Lambda := \Lambda \oplus \left[L^i_{y_{j^*}=0}, L^i_{y_{j^*}=1}\right]$. Aller à l'étape 2.

Étape 8 : Si $\Lambda \neq [\,]$, alors aller à l'étape 2; sinon, l'algorithme se termine, et:
si $z_{sup} < \infty$, alors la solution (x^*, w^*, y^*) satisfaisant $z_{sup} = z(x^*, w^*, y^*)$ est optimale;
si $z_{sup} = \infty$, alors le domaine réalisable est vide.

Il est à noter que la résolution de la relaxation faible du problème original permet de déterminer immédiatement si le domaine réalisable est vide ou non.

Dans la suite du texte, nous désignerons par "itération" chaque passage par l'étape 2 de l'algorithme 4.2.

4.2.6 Implantation

Nous avons réalisé une implantation en langage Fortran de l'algorithme 4.2.

La pile est implantée au moyen de tableaux résumant l'information nécessaire à la caractérisation d'un noeud de l'arbre. Ainsi, pour chaque noeud est emmagasiné un vecteur d'état des dépôts, précisant pour chaque dépôt si celui-ci est ouvert, fermé ou libre.

Ce vecteur est normalement remis à jour lors de la création des sous-problèmes mais il peut également être modifié par des procédures qui permettent de simplifier la résolution du problème. La première de ces procédures est appelée au début du programme, avant la résolution de la relaxation faible du problème original, et permet de fixer à 0 les dépôts qui ne sont pas connectés au réseau. La deuxième est appelée à chaque itération et permet de fixer à 1 les dépôts qui sont les seuls à pouvoir servir un client pour un produit donné. Ces deux procédures permettent d'éviter l'examen de sous-problèmes dont le domaine réalisable est vide.

La relaxation faible est résolue au moyen d'une version en nombres réels du logiciel RNET (Grigoriadis et Hsu, 1979). Ce dernier permet de résoudre un problème de flot à coût minimum par une méthode primale basée sur l'algorithme du simplexe. Ainsi, à chaque itération, le schéma suivant est implanté:

- $\bar{z} := 0$.
- Pour chaque produit p, faire:
  - construire le réseau;
  - résoudre un problème de flot à coût minimum sur le réseau ainsi construit;
  - soit $\overline{z_p}$ sa solution optimale; $\bar{z} := \bar{z} + \overline{z_p}$.

RNET permet également de recueillir les flots circulant sur les arcs du réseau. Ceux-ci peuvent servir dans le calcul du coefficient de la règle de branchement utilisée, de même que pour la sortie des résultats lorsque la solution identifiée est réalisable et donc candidate à l'optimalité.

Le choix de la règle de branchement, de même que l'utilisation de critères auxiliaires d'arrêt de l'algorithme peuvent être décidés par l'usager. Le premier critère permet l'arrêt de l'algorithme lorsque le nombre d'itérations exécutées atteint un maximum fixé par l'usager. Le deuxième critère force l'arrêt de l'algorithme lorsque l'écart relatif entre la meilleure borne supérieure identifiée (z) et la valeur optimale de la relaxation faible du problème original ($\overline{z_0}$) ne dépasse pas un certain seuil. Nous appelons cet écart relatif le *gap*. Formellement, l'algorithme se termine lorsque

$$gap = \frac{z - \overline{z_0}}{z} \leq \varepsilon,$$

$\varepsilon > 0$ fixé par l'usager.

CHAPITRE 5

IMPLANTATIONS PARALLÈLES

Les applications du parallélisme à des problèmes d'optimisation en recherche opérationnelle sont nombreuses. Zenios (1989) fournit une bibliographie détaillée sur les aspects numériques de l'optimisation parallèle. Kindervater et Lenstra (1988) font une revue de la littérature sur le traitement parallèle appliqué à des problèmes d'optimisation combinatoire. La méthode générale de séparation et évaluation progressive, présentée à la section 4.1, est justement utilisée pour résoudre des problèmes de cette classe. Nous consacrons ce chapitre aux applications du parallélisme à cette méthode et plus particulièrement à l'adaptation qui en a été faite pour résoudre le problème de localisation avec équilibrage.

Nous passons d'abord en revue la littérature sur les algorithmes parallèles de séparation et évaluation progressive. Nous discutons ensuite de certaines implantations parallèles de l'algorithme de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage.

5.1 Algorithmes parallèles de séparation et évaluation progressive

Nous pouvons identifier deux sources principales de parallélisme dans un algorithme de séparation et évaluation progressive:

- l'exploration parallèle de l'arbre de séparation et évaluation progressive;
- le calcul parallèle de la borne inférieure à chaque noeud de l'arbre.

Comme la deuxième source de parallélisme est dépendante du problème traité, nous ne nous intéressons dans cette section qu'à la première. Celle-ci consiste en l'évaluation ou l'examen en parallèle de plusieurs noeuds.

Un premier modèle d'exploration parallèle suppose que l'évaluation d'un noeud de l'arbre prend un temps constant et que tous les processeurs ont accès à une mémoire commune. Ainsi, la mise à jour de la borne supérieure et l'élimination des noeuds (par exemple, lorsque ceux-ci ont une borne inférieure au moins aussi grande que la meilleure borne supérieure connue) peuvent se faire en parallèle.

Avec un tel modèle, Burton, Huntbach, McKeown et Rayward-Smith (1983), donnent des exemples où l'utilisation de deux processeurs diminue de plus que la moitié le temps requis par un seul processeur, et d'autres où deux processeurs sont plus lents qu'un seul. Lai et Sahni (1984) étudient également de tels exemples d'anomalies mais montrent par une expérimentation numérique que leur fréquence d'apparition est faible. Lai et Sprague (1985) et Li et Wah (1986) font des études théoriques plus poussées de telles anomalies.

Les hypothèses du modèle présenté plus haut sont fortes. Ainsi, supposer un temps constant pour l'évaluation de chaque noeud entraîne une implantation où les calculs parallèles sont complètement synchronisés. Ceci peut conduire à des délais importants, susceptibles de nuire à la performance globale de l'algorithme. De plus, une mémoire commune est difficile à réaliser à cause de l'augmentation rapide des temps d'accès avec le nombre de processeurs (Athas et Seitz, 1988).

Dans la suite de cette section, nous nous intéressons donc uniquement à des implantations sur des systèmes multiprocesseurs à échanges de messages. Mentionnons que des implantations ont été réalisées sur des ordinateurs SIMD (Kindervater et Trienekens, 1988), sur une machine basée sur le modèle "dataflow" (Kindervater et Trienekens, 1988) et sur une architecture spécialement conçue pour résoudre des problèmes d'optimisation combinatoire (Wah et Ma, 1984).

Nous identifions deux approches principales pour l'exploration parallèle d'un arbre de séparation et évaluation progressive sur un système multiprocesseur à échanges de messages:

- *Approche collégiale*

  Dans cette approche, chaque processeur explore un sous-arbre. Chacun garde en mémoire locale sa liste de sous-problèmes générés mais non-examinés, de même

que sa propre valeur de la meilleure borne supérieure. Les processeurs se communiquent entre eux leurs bornes supérieures, de même que des noeuds non-examinés.

- *Approche maître-esclave.*
  Dans cette approche, un seul processeur, appelé maître, conserve une liste des sous-problèmes générés mais non-examinés, de même que la valeur de la meilleure borne supérieure connue. Les sous-problèmes à traiter sont partagés entre les processeurs. Les communications se font uniquement entre le maître et les autres processeurs, dits esclaves.

Les deux prochaines sous-sections sont consacrées aux implantations de chacune des ces approches, décrites dans la littérature.

5.1.1 Approche collégiale

Dessouki et Huen (1980) proposent un algorithme asynchrone de "branch-and-bound" divisé en trois phases principales:

- *Phase de sélection.*
  Dans cette phase, les processeurs choisissent un sous-arbre à explorer. Les mêmes noeuds peuvent être examinés simultanément par plusieurs processeurs.

- *Phase d'énumération.*
  Chaque processeur effectue sa propre recherche en profondeur dans son sous-arbre. Lorsqu'une nouvelle borne supérieure est identifiée, elle est distribuée à travers le réseau de processeurs.

- *Phase d'échange.*
  Lorsqu'un processeur p termine l'exploration de son sous-arbre, commence alors une phase d'échange dans laquelle:
  p doit identifier un processeur q auprès duquel il peut obtenir un noeud;
  q doit identifier un noeud à communiquer à p et mettre à jour son sous-arbre de recherche.

Dans cet algorithme, les communications se font au moyen de tâches de réception et d'envoi qui s'exécutent concurremment à la tâche principale.

Dessouki et Huen ne donnent aucun résultat numérique basé sur une implantation de leur algorithme.

Kindervater (1986) propose un algorithme parallèle de séparation et évaluation progressive que nous classons parmi les méthodes collégiales, bien qu'il s'exécute sur une architecture de type maître-esclave. En effet, dans cet algorithme, les esclaves font leur propre recherche dans un sous-arbre, mais ne peuvent communiquer directement entre eux. Les échanges d'informations passent tous par le maître.

Un processus maître génère l'arbre de recherche jusqu'à une certaine profondeur. Les noeuds générés mais non-examinés sont triés selon la qualité de leur borne inférieure. Le maître expédie alors ces noeuds à examiner à des processus esclaves. Ceux-ci commencent leur propre recherche en profondeur en prenant le noeud reçu comme racine. Les esclaves communiquent à intervalles réguliers avec le maître pour la mise à jour de la borne supérieure et l'envoi de noeuds vers des processus inoccupés.

Les résultats numériques montrent que lorsque le nombre d'esclaves augmente, le maître n'arrive pas à satisfaire toutes les demandes des esclaves. Ainsi, les performances de l'implantation plafonnent rapidement.

Anderson et Chen (1987) présentent un algorithme parallèle de séparation et évaluation progressive sur une architecture hypercube. L'idée maîtresse de leur algorithme est d'équilibrer la charge entre les processeurs. Dans ce but, chaque processeur calcule une mesure de la qualité de sa liste de sous-problèmes à examiner. Par exemple, pour une recherche en profondeur, la qualité de la liste peut être mesurée par la somme des profondeurs des noeuds du sous-arbre. Ainsi, les processeurs voisins communiquent régulièrement entre eux des informations sur la qualité de leur sous-arbre. S'il y a lieu, ils s'échangent alors des noeuds dans le but d'équilibrer leurs charges de travail. Cette phase de communication s'accompagne également d'échanges des bornes supérieures entre voisins. Un mécanisme de détection de la fin de l'algorithme est également décrit par les auteurs.

D'autres algorithmes collégiaux sont donnés par Pruul (1975) et Quinn (1990).

5.1.2 Approche maître-esclave

Trienekens (1986) suggère un algorithme parallèle de séparation et évaluation progressive basé sur une approche maître-esclave. Le maître conserve une liste des noeuds, qui sont triés selon la qualité de leur borne inférieure. Un esclave inoccupé reçoit un noeud, génère ses fils et calcule leurs bornes inférieures. Un avantage de cet algorithme est que le maître a pleine connaissance de l'arbre généré, contrairement à l'algorithme proposé par Kindervater (1986) et présenté à la section précédente. Mentionnons que si le temps de calcul de la borne inférieure est grand, comme c'est le cas pour le problème traité par Trienekens, il dominera le temps de communication.

Un autre algorithme de type maître-esclave est proposé par Quinn (1990). Mentionnons également une étude théorique des algorithmes de séparation et évaluation progressive de type maître-esclave, due à Boxma et Kindervater (1987). Cette étude est basée sur la théorie des files d'attente.

5.2 Implantations parallèles de l'algorithme de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage

Notre but est de développer des implantations parallèles de l'algorithme de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage, présenté à la section 4.2. Pour ce faire, nous utilisons l'environnement parallèle sur Transputers décrit au chapitre 2. Les logiciels de programmation sont des extensions du langage Fortran. Nous avons donc en vue la conception d'implantations parallèles efficaces qui utilisent une partie du code séquentiel déjà existant.

Comme pour un algorithme général de séparation et évaluation progressive, nous pouvons identifier deux sources principales de parallélisme:

- l'exploration parallèle de l'arbre de séparation et évaluation progressive;
- le calcul parallèle de la relaxation faible à chaque noeud de l'arbre.

Chacune de ces sources de parallélisme donne lieu à des implantations particulières, que nous étudions en détail dans la suite de cette section. En conclusion, nous nous interrogeons sur la portée des essais numériques à effectuer sur ces implantations.

5.2.1 Exploration parallèle de l'arbre

La conception d'un algorithme d'exploration parallèle est influencé par le choix d'une approche, collégiale ou maître-esclave, de même que par le choix de modèles de programmation parmi ceux vus au chapitre 2. Nous voyons tour à tour ces différents aspects avant d'étudier l'algorithme choisi et de traiter des détails de ses implantations.

5.2.1.1 Choix d'une approche

Une approche maître-esclave pour résoudre le problème de localisation avec équilibrage possède de nombreux avantages:

- Le temps élevé requis par la résolution de la relaxation faible à chaque noeud de l'arbre l'emporte nettement sur le temps des communications entre le maître et les esclaves.

- Le fait que l'arbre généré est entièrement connu par le maître permet un meilleur contrôle de l'exploration parallèle.

- Il est possible de réaliser des implantations qui soient indépendantes de la topologie du réseau d'interconnexion.

- Il est alors relativement facile de concevoir un algorithme synchrone. L'implantation d'un tel algorithme favorise la détection des erreurs de programmation.

Ce dernier aspect entraîne cependant un désavantage important, soit la perte de temps due à la synchronisation des communications. Mentionnons également le fait que les communications passent toutes par le maître, ce qui peut rendre cette approche peu intéressante lorsque le nombre de processeurs est élevé.

L'approche collégiale tente d'occuper les processeurs au maximum et de minimiser le temps de communication. Cependant, elle comporte quelques désavantages:

- Il est difficile de contrôler l'exploration globale de l'arbre puisque chaque processeur traite son propre sous-arbre.

- Il peut être difficile de réaliser une implantation qui demeure efficace quelque soit la topologie du réseau d'interconnexion.

- Un algorithme asynchrone est approprié pour une telle approche; cependant, il rend difficile la détection des erreurs de programmation.

Comme nous utilisons un nombre relativement peu élevé de processeurs (16 et moins) et que le calcul de la relaxation faible demande un temps considérable, nous jugeons que l'approche maître-esclave est plus appropriée pour résoudre le problème de localisation avec équilibrage sur un réseau de Transputers.

Il reste cependant quelques points à éclaircir dans la conception de l'algorithme.

Le premier est le moment du début de l'exploration parallèle. Il peut être désavantageux de commencer l'exploration parallèle au tout début. Lors de l'exploration séquentielle, les noeuds non-examinés au début et qui se trouvent près de la racine de l'arbre, sont en général peu prometteurs. Une exploration parallèle efficace gagnera donc à retarder leur examen le plus possible.

Un deuxième aspect à éclaircir est le tri de la liste des sous-problèmes générés mais non-examinés. L'exploration séquentielle ordonne les sous-problèmes selon leur profondeur. Un tel critère peut être adapté à l'exploration parallèle en choisissant d'évaluer en priorité un des noeuds qui se trouvent à plus grande distance de la racine.

5.2.1.2 Choix de modèles de programmation

Comme nous l'avons vu au chapitre 2, le logiciel Parallel Fortran donne au programmeur deux options: la programmation directe et la technique "processor farms".

La programmation directe donne plein contrôle au programmeur sur les échanges d'informations entre les processeurs. Cependant, à moins de programmer soi-même une tâche auxiliaire de communication, les échanges ne peuvent avoir lieu qu'entre des processeurs ayant un lien physique entre eux. Également, le processeur Racine étant le seul à pouvoir appeler des fonctions d'E/S, il doit être le processeur maître. Puisque

chaque Transputer est doté de quatre liens, si le programmeur n'utilise que les liens physiques directs pour la communication maître-esclave, la programmation directe nous restreint à n'utiliser que trois esclaves (l'autre lien utilisé étant le lien Racine-Hôte).

La technique "processor farms" permet d'utiliser un nombre arbitraire d'esclaves. Cependant, le programmeur n'a aucun contrôle sur la destination des messages envoyés par le maître; c'est la tâche Frouter (voir section 2.3.1.2) qui prend la décision pour lui.

Une façon efficace d'implanter un algorithme synchrone de type maître-esclave par la technique "processor farms" est donnée par Chabini (1990). À chaque étape de communication, le maître répond à la demande d'un esclave anonyme qui requiert un paquet de données à traiter ou désire envoyer des résultats. Le choix de l'esclave qui communique ainsi avec le maître est entièrement laissé à la tâche Frouter.

Lorsque le maître reçoit des résultats d'un esclave et qu'il reste des paquets à traiter, il lui en envoie un immédiatement. Si certains esclaves traitent rapidement les paquets reçus, il est alors possible qu'une fraction des esclaves ne travaillent jamais.

Par contre, avec la programmation directe, il est facile d'implanter un algorithme synchrone de type maître-esclave dans lequel tous les esclaves sont occupés (en supposant que le nombre de paquets est supérieur ou égal au nombre d'esclaves). L'algorithme est le suivant: tant qu'il reste un paquet à traiter et qu'il y a au moins un esclave libre, le maître expédie un paquet. Ensuite, jusqu'à ce que tous les paquets aient été envoyés et les résultats reçus, le maître vérifie périodiquement si un esclave veut communiquer avec lui. Dans l'affirmative, il reçoit les résultats de l'esclave, puis, s'il reste au moins un paquet à traiter, lui en envoie un.

Voici un exemple illustrant les constatations que nous venons de faire. Nous disposons de trois esclaves, notés E_i (i=1,2,3) et nous avons quatre paquets de données à traiter, notés P_j (j=1,2,3,4). Nous supposons que le temps de traitement du paquet P_j est donné par $t(P_j) = j$, et que le temps de communication maître-esclave, t_c , est unitaire ($t_c = 1$), qu'il s'agisse d'un paquet à expédier ou de résultats à recevoir. À chaque étape de l'exécution, chaque esclave peut être dans un des quatre états suivants:

- il reçoit un paquet à traiter;
- il traite un paquet;
- il envoie des résultats au maître;

- il attend que le maître lui réponde.

Le tableau suivant résume l'exécution par les deux approches, programmation directe et "processor farms". La colonne à gauche ("t") indique les étapes de l'exécution. Chacune des quatre colonnes sous "processor farms" et "programmation directe", indique un des états décrits ci-haut. La notation E_i:P_j indique qu'un esclave E_i est en charge du paquet P_j, quelque soit l'état dans lequel se trouve E_i.

t	"processor farms"				programmation directe			
	REC.	TRAV.	ENV.	ATT.	REC.	TRAV.	ENV.	ATT.
1	E_1:P_1				E_1:P_1			
2	E_2:P_2	E_1:P_1			E_2:P_2	E_1:P_1		
3		E_2:P_2	E_1:P_1		E_3:P_3	E_2:P_2		E_1:P_1
4	E_1:P_3	E_2:P_2				E_2:P_2 E_3:P_3	E_1:P_1	
5		E_1:P_3	E_2:P_2		E_1:P_4	E_3:P_3		E_2:P_2
6	E_2:P_4	E_1:P_3				E_3:P_3 E_1:P_4	E_2:P_2	
7		E_1:P_3 E_2:P_4				E_1:P_4	E_3:P_3	
8		E_2:P_4	E_1:P_3			E_1:P_4		
9		E_2:P_4				E_1:P_4		
10		E_2:P_4					E_1:P_4	
11			E_2:P_4					

Tableau I. Comparaison de la programmation directe et de la technique "processor farms", sur un exemple

Comme le montre l'exemple, la technique "processor farms" ne permet pas nécessairement d'utiliser tous les esclaves disponibles. Il s'ensuit une perte d'efficacité de la méthode par rapport à la programmation directe, où le programmeur peut décider d'utiliser tous les esclaves disponibles.

Les résultats obtenus par Chabini (1990) confirment notre analyse. Chabini a réalisé des implantations parallèles de l'algorithme d'approximation linéaire pour résoudre le problème d'affectation du trafic, en utilisant la programmation directe et la technique "processor farms". Les temps d'exécution obtenus par cette dernière sont nettement moins bons que ceux obtenus par la programmation directe.

Cette dernière approche peut d'ailleurs se généraliser à plus de quatre processeurs, par l'utilisation du logiciel Express. L'avantage de ce dernier est qu'il permet une communication "lâchement synchronisée" (voir section 2.3.2.3) entre n'importe quelle paire de processeurs. Contrairement à la programmation directe de Parallel Fortran, il est donc possible d'implanter un algorithme synchrone de type maître-esclave avec un nombre arbitraire d'esclaves. Dans ce sens, une telle implantation est indépendante du réseau d'interconnexion.

Express permet deux types de programmation possibles: l'approche "host-node" et l'approche Cubix.

La première requiert un programme maître fonctionnant sur le processeur Hôte (le PC). Or, pour permettre une comparaison plus exacte des performances des implantations séquentielles et parallèles, nous avons développé l'algorithme séquentiel sur un Transputer, en utilisant le compilateur Fortran de 3L. Nous adoptons donc une approche Cubix, qui n'utilise que des programmes fonctionnant sur les Transputers.

5.2.1.3 Algorithme

La procédure maître est divisé en deux phases: une phase d'exploration séquentielle et une phase d'exploration parallèle. Les deux phases sont différenciées par une variable booléenne D, telle que

$$D = \begin{cases} VRAI, & \text{si l'algorithme est dans la phase parallèle} \\ FAUX, & \text{sinon} \end{cases}$$

La phase séquentielle se poursuit jusqu'à ce que le nombre d'itérations, représenté par la variable *iter*, atteigne une valeur N, déjà fixée. Commence alors une étape de distribution des sous-problèmes générés mais non-examinés, à travers le réseau de processeurs. Le maître garde un sous-problème à traiter et expédie aux esclaves d'autres sous-problèmes, tant qu'il en reste dans la pile ou que chaque esclave en a reçu un.

Le nombre d'esclaves, $NESC$, est fixé à l'avance alors que le nombre d'esclaves actifs est désigné par la variable $nesc_a$.

Lors de la phase d'exploration parallèle, la variable *sync* permet de synchroniser le traitement d'un sous-problème par le maître avec la réception des résultats et l'envoi de sous-problèmes. La variable *sync* varie entre 0 et $nesc_a$.

Lorsque *sync* vaut 0, le maître tente de résoudre un sous-problème et il faut alors distinguer deux cas. Si la pile n'est pas vide, le maître traite un sous-problème. Si la pile est vide, le maître recueuille les résultats venant d'un esclave.

Lorsque *sync* est supérieur à 0, le maître reçoit les résultats d'un esclave p et alors deux cas se présentent. Si la pile n'est pas vide, le maître expédie un sous-problème à l'esclave p. Si la pile est vide, la variable $nesc_a$ est décrémentée, puisque l'esclave p devient inactif.

À chaque itération (résolution d'un sous-problème par le maître ou réception de résultats venant d'un esclave), la variable *sync* est incrémentée modulo ($nesc_a + 1$).

Enfin, lors de la phase d'exploration parallèle, une dernière étape est ajoutée afin de permettre au plus grand nombre possible d'esclaves d'être actifs en même temps. Elle consiste à envoyer aux esclaves inactifs des sous-problèmes, tant qu'il en reste dans la pile .

L'algorithme se termine lorsque la pile est vide et qu'il n'y a plus aucun esclave actif. Le maître signale alors aux esclaves la fin de l'algorithme.

Nous résumons ci-dessous les procédures maître et esclave de l'algorithme parallèle de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage. Les notations autres que celles introduites plus haut sont les mêmes que celles utilisées dans l'algorithme 4.2 du chapitre 4.

Algorithme 5.1: Procédure maître de l'algorithme parallèle de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage.

Étape 1 : $\Lambda := [\,]$; $z_{sup} := \infty$; $iter := 0$; $D := FAUX$; $nesc_e := 0$.

Étape 2 : Si ($D = VRAI$), alors:

Etape 2.1 : Si ($sync = 0$ et $\Lambda \neq [\,]$), alors:
$\Lambda := \Lambda \ominus [L_i]$;
résoudre la relaxation faible R_i de L_i .

Étape 2.2 : Si (($sync = 0$ et $\Lambda = [\,]$) ou $sync > 0$), alors:
recevoir les résultats d'un esclave p sur la relaxation faible R_i d'un problème L_i .

Étape 2.2.1 : Si ($\Lambda \neq [\,]$), alors:
$\Lambda := \Lambda \ominus [L_e]$;
envoyer à l'esclave p le problème L_e .

Étape 2.2.2 : Si ($\Lambda = [\,]$), alors:

$nesc_a := nesc_a - 1$;

si ($sync = nesc_a + 1$), alors $sync := nesc_a - 1$.

Étape 2.3 : $sync := (sync + 1)$ MOD $(nesc_a + 1)$.

Étape 2.4 : Si ($nesc_a < NESC$), alors:

Tant que ($\Lambda \neq [\,]$ et $nesc_a < NESC$), faire:

$\Lambda := \Lambda \ominus [L_a]$;

envoyer à un esclave inactif le problème L_a ;

$nesc_a := nesc_a + 1$.

Étape 3 : Si ($D = FAUX$), alors:

Étape 3.1 : Si ($iter \neq N$), alors:

$\Lambda := \Lambda \ominus [L_i]$;

résoudre la relaxation faible R_i de L_i ;

$iter := iter + 1$.

Étape 3.2 : Si ($iter = N$), alors:

$\Lambda := \Lambda \ominus [L_i]$;

Tant que ($\Lambda \neq [\,]$ et $nesc_a < NESC$), faire:

$\Lambda := \Lambda \ominus [L_a]$;

envoyer à un esclave inactif le problème L_a;

$nesc_a := nesc_a + 1$.

Si ($nesc_a > 0$), alors:

$D := VRAI$;

$sync := 1$.

résoudre la relaxation faible R_i de L_i .

Étape 4 : Si $D(R_i) = \varnothing$, alors aller à l'étape 9.

Étape 5 : Soient $(\bar{x}, \bar{w}, \bar{y})$, solution optimale de R_i et $\bar{z}$, sa valeur optimale. Si $(\bar{x}, \bar{w}, \bar{y}) \in D(L_i)$, alors:
si $\bar{z} < z_{sup}$, alors $z_{sup} := \bar{z}$;
aller à l'étape 9.

Étape 6 : Si $\bar{z} \geq z_{sup}$, alors aller à l'étape 9.

Étape 7 : Soit $z' := z(\bar{x}, \bar{w}, y')$, où

$$y'_j = \begin{cases} 1, & \text{si } \bar{y}_j > 0 \\ 0, & \text{sinon} \end{cases}$$

Si $z' < z_{sup}$, alors $z_{sup} := z'$.

Étape 8 : Choisir une variable y_{j^*}, $j^* \in \overline{K_{01}}$, et créer les deux sous-problèmes $L^i_{y_{j^*}=0}, L^i_{y_{j^*}=1}$. Insérer les deux sous-problèmes dans la pile de telle sorte que celle-ci soit ordonnée selon la profondeur.

Étape 9 : Si ($\Lambda \neq [\,]$ ou $nesc_a > 0$), alors aller à l'étape 2; sinon, l'algorithme se termine et:
si $z_{sup} < \infty$, alors (x^*, w^*, y^*) satisfaisant $z_{sup} = z(x^*, w^*, y^*)$ est optimale;
si $z_{sup} = \infty$, alors le domaine réalisable est vide;
signifier à tous les esclaves la fin de l'algorithme.

Algorithme 5.2: Procédure esclave de l'algorithme parallèle de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage.

Étape 1 : Recevoir du maître un problème L_i ou le signal de la fin de l'algorithme. Dans ce dernier cas, l'algorithme se termine.

Étape 2 : Résoudre la relaxation faible R_i de L_i.

Étape 3 : Envoyer au maître les résultats de la résolution de R_i .

5.2.1.4 Implantations

Nous avons réalisé deux implantations des algorithmes 5.1 et 5.2. La première utilise l'approche de programmation directe (section 2.3.1.1) du logiciel Parallel Fortran, alors que la deuxième est basée sur l'approche Cubix (section 2.3.2.2) du logiciel Express.

Ces deux implantations ne diffèrent que par la localisation du processeur maître et le nombre maximum de processeurs esclaves utilisés. Ainsi, avec l'approche de programmation directe, le processeur maître est le Transputer Racine et le nombre maximum de processeurs esclaves est trois. Avec l'approche Cubix, le processeur maître peut être quelconque puisque les fonctions d'E/S sont accessibles à partir de n'importe quel processeur. De plus, avec cette approche, le nombre maximum de processeurs esclaves est illimité.

Les autres caractéristiques sont les mêmes pour chacune des deux implantations. Mentionnons-en quelques-unes.

Une phase préliminaire de communications est nécessaire pour distribuer les données du problème aux esclaves.

D'autre part, un sous-problème à un noeud donné de l'arbre, est entièrement caractérisé par le vecteur d'état des dépôts. C'est donc uniquement ce vecteur qui est communiqué lors de l'envoi d'un sous-problème par le maître.

Les résultats envoyés au maître par un esclave sont, d'abord, le critère de réalisabilité de la relaxation faible traitée (étape 4 de l'algorithme 5.1), puis s'il y a lieu, la solution optimale $(\overline{x}, \overline{w}, \overline{y})$ et sa valeur optimale $\overline{z}$.

Le tri de la pile selon la profondeur, lors de l'insertion de deux nouveaux sous-problèmes à l'étape 8, est réalisé de la façon suivante. La profondeur des nouveaux sous-problèmes générés est calculée, puis une recherche dichotomique d'un noeud ayant

la même profondeur est effectuée. Lorsque ce noeud est trouvé, tous les noeuds qui lui sont supérieurs dans la pile sont déplacés de deux espaces. Les deux espaces vacants sont alors occupés par les deux nouveaux sous-problèmes générés.

Les critères auxiliaires d'arrêt, mentionnés à la section 4.2.6, sont également disponibles dans les implantations parallèles. Ainsi, lorsque le nombre maximum d'itérations (déterminé par l'usager) moins le nombre d'esclaves actifs ($nesc_a$), est atteint, ou que le critère de *gap*, choisi par l'usager, est satisfait, le maître reçoit les $nesc_a$ sous-problèmes traités par les esclaves. Il signale ensuite à ceux-ci la fin de l'algorithme.

5.2.2 Calcul parallèle de la relaxation faible

Nous pouvons identifier deux sources principales de parallélisme dans la résolution de la relaxation faible à chaque noeud de l'arbre de séparation et évaluation progressive:

- la résolution parallèle des l problèmes de flot à coût minimum mutuellement indépendants;
- la résolution parallèle d'un problème de flot à coût minimum.

Dans le cadre de ce travail, nous nous sommes restreints à la première possibilité.

Le problème consiste à distribuer les problèmes de flot à coût minimum à travers l'ensemble des processeurs, de façon telle que la charge soit la plus équilibrée possible. Pour cela, nous faisons l'hypothèse simplificatrice que les l problèmes de flot à coût minimum demandent des temps de résolution comparables.

Supposons qu'il y a *NPROC* processeurs. Définissons alors

$$q = l \quad DIV \quad NPROC$$

$$r = l \quad \text{MOD} \quad NPROC$$

où *DIV* est la division entière et MOD donne le reste de cette division. Alors, r processeurs vont recevoir chacun $(q + 1)$ problèmes à résoudre et les $(NPROC - r)$ processeurs restants vont chacun avoir q problèmes à résoudre.

Ce schéma de distribution est facilement implanté au moyen d'un algorithme maître-esclave, où le maître est un des processeurs ayant q problèmes à résoudre. Le nombre de sous-problèmes à résoudre par chaque processeur est déterminé par le maître au début de l'exécution et demeure le même à chaque résolution d'une relaxation faible.

Le schéma suivant est implanté pour le maître:

- Envoyer $(l - q)$ problèmes à résoudre aux ($NPROC$ - 1) processeurs esclaves.
- $\bar{z} := 0$.
- Pour chaque produit p à résoudre, faire:
  - construire le réseau;
  - résoudre un problème de flot à coût minimum sur le réseau ainsi construit;
  - soit $\bar{z}_p$ sa solution optimale; $\bar{z} := \bar{z} + \bar{z}_p$.
- Pour i:= 1 à ($NPROC$ - 1), faire:
  - recevoir le résultat $\bar{z}_e$ d'un processeur esclave;
  - $\bar{z} := \bar{z} + \bar{z}_e$.

Le schéma suivant est implanté pour l'esclave:

- Recevoir q ou $(q + 1)$ problèmes à résoudre.
- $\bar{z}_e := 0$.
- Pour chaque produit p à résoudre, faire:
  - construire le réseau;
  - résoudre un problème de flot à coût minimum sur le réseau ainsi construit;
  - soit $\bar{z}_p$ sa solution optimale; $\bar{z}_e := \bar{z}_e + \bar{z}_p$.
- Envoyer le résultat $\bar{z}_e$ au maître.

Nous avons réalisé deux implantations de ces schémas. La première est basée sur l'approche de programmation directe de Parallel Fortran, alors que la deuxième utilise l'approche Cubix du logiciel Express.

5.2.3 Conclusion

Nous terminons cette section en dégageant quelques interrogations auxquelles nous tenterons de répondre par des essais numériques.

Rappelons que l'objectif de l'exploration parallèle de l'arbre consiste non seulement à diminuer substantiellement le temps d'exécution, par rapport à une exploration séquentielle, mais également à obtenir des solutions au moins aussi bonnes, en un nombre comparable d'itérations.

Ayant en vue cet objectif, nous tenterons de répondre aux questions suivantes:

- À quel moment doit commencer l'exploration parallèle? Autrement dit, quelle doit être la durée de la phase d'exploration séquentielle? (paramètre N de l'algorithme 5.1)

- Comment varient les performances de l'exploration parallèle en fonction du nombre de processeurs?

- Comment varient les performances de l'exploration parallèle en fonction de la complexité des problèmes traités?

- Le tri de la liste des problèmes générés mais non-examinés selon le critère de la profondeur, est-il efficace?

- Comment se comparent les performances des implantations avec Parallel Fortran et avec Express?

- Comment relier entre eux les processeurs de façon à minimiser le temps des communications?

Nous nous interrogerons également sur les performances du calcul parallèle de la relaxation faible en fonction du nombre de processeurs utilisés et de la complexité (en particulier, le nombre de produits) des problèmes traités.

Remarquons que nous n'avons pas réalisé d'implantations qui permettent de combiner les deux sources principales de parallélisme en un seul algorithme. La conception de cet algorithme peut faire l'objet de recherches ultérieures, basées sur les résultats numériques que nous présentons au prochain chapitre.

CHAPITRE 6

RÉSULTATS NUMÉRIQUES

Ce chapitre est consacré aux résultats numériques des tests effectués sur l'implantation séquentielle, décrite au chapitre 4, et sur les implantations parallèles, décrites au chapitre 5. Les tests ont été menés sur une série de problèmes générés aléatoirement, de même que sur des données provenant d'une application réelle.

Rappelons que le but de ces tests est de mesurer les performances des deux principales sources de parallélisme dans l'algorithme de séparation et évaluaion progressive: l'exploration parallèle de l'arbre et le calcul parallèle de la relaxation faible.

Les performances de l'exploration parallèle dépendent d'une série de facteurs, dont:

- la durée de la phase d'exploration séquentielle;
- l'efficacité du tri selon la profondeur, des problèmes générés mais non-examinés;
- le nombre de processeurs;
- la complexité des problèmes traités;
- la topologie du réseau d'interconnexion.

Ces trois derniers facteurs jouent également un rôle important dans le calcul parallèle de la relaxation faible. Nous pouvons aussi nous demander si l'hypothèse simplificatrice que les l problèmes de flot à coût minimum demandent des temps de résolution comparables, tient pour tous les types de problèmes traités.

Ce chapitre est divisé en cinq sections. Nous voyons d'abord en détail les caractéristiques des problèmes testés. Par la suite, nous identifions des paramètres permettant de mesurer les performances des implantations. Finalement, nous donnons les résultats

obtenus avec les problèmes générés aléatoirement, puis ceux obtenus avec les données de l'application réelle. En conclusion, nous portons un jugement global sur les résultats obtenus.

6.1 Problèmes testés

Les problèmes générés aléatoirement sont caractérisés principalement par:

- le nombre de noeuds (clients et dépôts);
- le nombre de zones dans le réseau;
- le nombre de produits.

Dans ce travail, nous nous restreignons à tester des problèmes dits moyens, caractérisés par un nombre de noeuds égal à 150. Ces problèmes sont notés MZ#P?, où # est le nombre de zones, qui peut prendre les valeurs 1,2 ou 4, et ? représente le nombre de produits qui peut prendre les valeurs 1 à 6.

Chacun des 18 problèmes ainsi générés possède 25 ou 26 dépôts à localiser, 125 ou 124 clients et de 2350 à 14124 arcs. Pour tous les problèmes, le nombre d'arcs associés à un produit donné est d'environ 2350. La procédure de génération aléatoire des données est décrite dans Crainic, Delorme et Dejax (1989).

L'application réelle provient des données d'une compagnie européenne de transport de conteneurs. Le réseau testé possède 12 produits, 90 dépôts, 289 clients et 37708 arcs.

6.2 Mesures de performance

Pour mesurer les performances des différentes implantations, nous introduisons trois paramètres:

- le *gap* de la meilleure solution trouvée;
- le nombre d'itérations nécessaires avant de trouver la meilleure solution;
- le temps d'exécution.

Le *gap*, déjà introduit au chapitre 4, représente l'écart relatif entre la meilleure solution réalisable identifiée (z) et la solution de la relaxation faible du problème original ($\overline{z_0}$), c'est-à-dire

$$gap = \frac{z - \overline{z_0}}{z}$$

Le *gap* sert à la fois à différencier entre elles les meilleures solutions trouvées par plusieurs implantations et à mesurer la qualité de la borne inférieure calculée par la relaxation faible.

Le nombre d'itérations nécessaires avant de trouver la meilleure solution, notée *NITER*, mesure la vitesse à laquelle l'implantation détermine une "bonne" solution. Nous n'avons pas retenu comme autre critère le nombre total d'itérations effectuées, puisque généralement, l'arbre de séparation et évaluation progressive n'est pas entièrement exploré.

Le temps d'exécution de l'implantation séquentielle est mesuré par le temps absolu en secondes. Le temps d'exécution des implantations parallèles est comparé à celui de l'implantation séquentielle au moyen de la notion d'**accélération**. Celle-ci, notée $a(NPROC)$ dépend du nombre de processeurs *NPROC* et est défini comme le rapport du temps d'exécution de l'implantation séquentielle à celui de l'implantation parallèle à *NPROC* processeurs.

6.3 Problèmes générés aléatoirement

Dans cette section, nous passons en revue les résultats des tests effectués sur les problèmes générés aléatoirement. Nous voyons d'abord les résultats de l'implantation séquentielle, puis ceux de l'exploration parallèle de l'arbre de séparation et évaluation progressive et enfin, ceux du calcul parallèle de la relaxation faible à chaque noeud de l'arbre.

6.3.1 Implantation séquentielle

L'implantation séquentielle a été réalisée en langage Fortran. Le code exécutable, créé avec le compilateur de 3L, est adapté au fonctionnement sur un Transputer.

Nous avons testé les 18 problèmes générés aléatoirement en utilisant comme règle de branchement la règle du quotient simplifié avec seuil M (voir section 4.2.4). Nous avons fixé un maximum de 1000 itérations; ce maximum a été atteint dans tous les problèmes, sans que l'arbre ait été entièrement exploré.

Le tableau II résume les réultats obtenus par l'implantation séquentielle. Pour chaque problème, le tableau fournit dans l'ordre, le *gap*, *NITER* et le temps total d'exécution en secondes.

	P1	P2	P3	P4	P5	P6
MZ1	12.4 403 684.08	22.2 120 1104.04	12.0 364 2039.57	7.1 34 2829.67	6.5 441 4032.84	3.1 676 5190.84
MZ2	31.9 352 538.50	19.2 879 1079.61	8.4 218 1644.08	7.0 539 2795.17	7.7 39 3439.16	3.5 619 4051.52
MZ3	29.4 433 760.87	18.9 501 1329.89	8.8 34 2112.88	10.1 30 2955.86	5.7 37 3758.16	4.9 217 4739.34

Tableau II. Performances de l'implantation séquentielle

6.3.2 Exploration parallèle de l'arbre

Nous étudions tour à tour les résultats des deux implantations, celle utilisant la programmation directe de Parallel Fortran et celle utilisant l'approche Cubix d'Express. Dans tous les tableaux de résultats rapportés dans cette section, nous marquerons par les symboles "+" et "-" les problèmes pour lesquels l'exploration parallèle a donné respectivement une meilleure et une moins bonne solution que l'exploration séquentielle.

6.3.2.1 Parallel Fortran

Les tests ont été effectués sur un Quadputer doté de quatre Transputers. La vitesse des liens de communication a été fixée à 10 mégabits/seconde.

Tous les problèmes générés aléatoirement ont été testés en utilisant la règle du quotient simplifié avec seuil M. Dans tous les essais, un maximum fixé à 1000 itérations a été atteint sans que l'arbre ait été complètement exploré.

Nous tentons d'abord de mesurer l'effet de la durée de la phase d'exploration séquentielle, en faisant varier le paramètre N (voir algorithme 5.1), qui mesure le nombre d'itérations séquentielles avant le début de la phase d'exploration parallèle. Nous étudions ensuite la variation des performances en fonction du nombre de processeurs. Finalement, nous nous intéressons à la question suivante: si nous éliminons le tri selon la profondeur de la liste des problèmes générés mais non-examinés, quelles performances obtenons-nous? Nous visons donc ici à mesurer l'efficacité du tri selon la profondeur.

Nous étudions tour à tour ces trois expériences dans les prochaines sous-sections.

6.3.2.1.1 Durée de l'exploration séquentielle

Rappelons que le but de la phase d'exploration séquentielle est d'éviter que l'exploration parallèle perde du temps à explorer des portions d'arbre peu prometteuses.

Nous choisissons de fixer le nombre de processeurs utilisés ($NPROC$) au maximum permis par l'implantation sur Parallel Fortran, soit quatre. Nous faisons varier le paramètre N qui mesure la durée de la phase d'exploration séquentielle. Pour déterminer

des valeurs à tester, rappelons que nous voulons étudier le comportement de l'exploration parallèle lorsqu'elle commence dans des régions profondes, peu profondes et intermédiaires de l'arbre. Ainsi, la valeur N=24 correspond, dans tous les problèmes, à une région où une première solution réalisable est près d'être identifiée. C'est donc une valeur appropriée pour tester une exploration parallèle qui commence dans une région profonde. À l'autre extrême, nous choisissons la valeur N=6. Les autres valeurs testées (9,12,15,18 et 21) sont distribuées également entre les deux extrêmes.

Les tableaux III à IX résument les résultats obtenus par ces tests. Pour chaque problème, les résultats rapportés sont le *gap* de la meilleure solution trouvée et l'itération où cette solution a été identifiée (*NITER*). Nous n'incluons pas les accélérations car nous visons uniquement à mesurer la qualité des solutions trouvées.

	P1	P2	P3	P4	P5	P6
MZ1	12.4 945	21.9^{+} 500	12.0 903	6.9^{+} 109	6.5 633	3.2^{-} 83
MZ2	30.9^{+} 997	19.5^{-} 257	8.4 554	7.3^{-} 88	7.5^{+} 95	- -
MZ4	30.0^{-} 962	18.9 958	8.8 103	10.1 122	5.7 85	4.9 274

Tableau III. Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC* = 4, N = 6)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 515	22.2 379	12.2* 992	7.1 93	6.5 585	3.1 77
MZ2	32.0* 466	19.3* 808	8.4 587	7.0 75	7.7 77	3.5 557
MZ4	29.4 792	18.9 621	8.8 90	10.1 93	5.7 87	4.9 329

Tableau IV. Performances de l'exploration parallèle, avec Parallel Fortran ($NPROC = 4, N = 9$)

	P1	P2	P3	P4	P5	P6
MZ1	12.5* 214	22.2 391	12.2* 457	6.9+ 357	6.5 970	3.1 719
MZ2	31.9 633	19.3* 719	8.4 498	7.0 600	7.7 100	3.5 661
MZ4	29.4 604	18.9 910	8.8 74	10.1 83	5.7 71	4.9 391

Tableau V. Performances de l'exploration parallèle, avec Parallel Fortran ($NPROC = 4, N = 12$)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 541	22.2 288	12.2⁻ 388	6.9⁺ 294	6.5 341	3.1 677
MZ2	31.9 566	19.2 153	8.4 558	7.0 609	7.7 57	3.5 656
MZ4	29.4 826	18.9 972	8.8 61	10.1 71	5.7 77	4.9 134

Tableau VI. Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC* = 4, *N* = 15)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 577	22.2 134	12.0 529	6.9⁺ 373	6.5 407	3.2⁻ 48
MZ2	31.9 657	19.2 662	8.4 181	7.0 587	7.7 98	3.5 291
MZ4	29.4 521	18.7⁺ 941	8.8 51	10.1 57	5.7 73	4.9 227

Tableau VII. Performances de l'exploration parallèle, avec Parallel Fortran (*NPROC* = 4, *N* = 18)

	P1	P2	P3	P4	P5	P6
MZ1	12.5* 752	22.2 283	12.0 334	6.9* 133	6.5 450	3.1 667
MZ2	31.9 709	18.8* 856	8.4 641	7.0 605	7.7 38	3.5 545
MZ4	29.4 337	18.9 555	8.8 52	10.1 56	5.7 63	4.9 265

Tableau VIII. Performances de l'exploration parallèle, avec Parallel Fortran ($NPROC = 4, N = 21$)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 598	22.2 122	12.0 451	6.9* 104	6.5 444	3.1 687
MZ2	31.9 374	19.3* 990	8.4 681	7.0 608	7.7 48	3.5 301
MZ4	29.0* 982	18.9 522	8.8 29	10.1 34	5.7 37	4.9 259

Tableau IX. Performances de l'exploration parallèle, avec Parallel Fortran ($NPROC = 4, N = 24$)

Les résultats montrent que pour N petit (6,9,12), les solutions trouvées diffèrent souvent sensiblement de celles obtenues par l'exploration séquentielle. Pour $N = 6$, il y a même un cas, identifié par des " - ", où aucune solution réalisable n'a été identifiée en moins de 1000 itérations. Pour N moyen ou grand (15, 18, 21, 24), les solutions trouvées se rapprochent des solutions séquentielles.

En particulier, la valeur $N = 24$ est intéressante, car, comme nous l'avons mentionné plus haut, elle correspond dans tous les problèmes à une région de l'arbre où une première solution réalisable est près d'être identifée. Nous voyons également que pour cette valeur de N, lorsque les solutions parallèle et séquentielle coïncident, le nombre d'itérations pour les atteindre est comparable.

Sur la base de ces résultats, nous pouvons donc suggérer une règle qui allie à la fois simplicité et robustesse:

> "L'exploration parallèle commence lorsque la première solution réalisable est atteinte."

6.3.2.1.2 Variation du nombre de processeurs

Notre but est ici de mesurer l'effet de l'augmentation du nombre de processeurs sur les performances de l'exploration parallèle. Le nombre de processeurs, *NPROC*, varie de 2 à 4 alors que le paramètre N est fixé à 24.

Les tableaux X à XII résument les résultats obtenus. Pour chaque problème, les tableaux fournissent dans l'ordre, le *gap*, *NITER* et l'acccélération $a(NPROC)$, calculée par rapport au temps total d'exécution.

	P1	P2	P3	P4	P5	P6
MZ1	12.4 574 1.31	22.2 121 1.30	12.0 411 1.33	7.1 35 1.35	6.5 458 1.37	3.2* 29 1.35
MZ2	31.9 357 1.28	19.2 813 1.32	8.4 212 1.32	7.0 554 1.35	7.7 39 1.34	3.5 271 1.36
MZ4	29.4 429 1.31	18.9 686 1.33	8.8 35 1.35	10.1 54 1.36	5.7 42 1.36	4.9 240 1.37

Tableau X. Performances de l'exploration parallèle, avec Parallel Fortran ($NPROC = 2, N = 24$)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 336 1.79	22.2 127 1.79	12.0 452 1.84	6.9+ 950 1.90	6.5 434 1.91	3.1 711 1.92
MZ2	31.9 349 1.74	19.2 769 1.81	8.4 205 1.82	7.0 571 1.87	7.7 43 1.88	3.5 548 1.89
MZ4	29.4 462 1.81	18.9 698 1.85	8.8 36 1.88	10.1 42 2.00	5.7 39 1.92	4.9 254 1.91

Tableau XI. Performances de l'exploration parallèle, avec Parallel Fortran ($NPROC = 3, N = 24$)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 598 2.18	22.2 122 2.26	12.0 451 2.38	6.9* 104 2.34	6.5 444 2.46	3.1 687 2.45
MZ2	31.9 374 2.24	19.3* 990 2.30	8.4 681 2.46	7.0 608 2.45	7.7 48 2.41	3.5 301 2.43
MZ4	29.0* 982 2.39	18.9 522 2.38	8.8 29 2.43	10.1 34 2.58	5.7 37 2.48	4.9 259 2.47

Tableau XII. Performances de l'exploration parallèle, avec Parallel Fortran ($NPROC = 4, N = 24$)

L'analyse des résultats montre que quelque soit le nombre de processeurs, les solutions obtenues sont similaires à celles trouvées par l'implantation séquentielle.

Pour ce qui est des temps d'exécution, nous remarquons les faits suivants:

- L'accélération augmente constamment en fonction du nombre de processeurs.
- Cette augmentation n'est pas linéaire; ce résultat est attendu, puisque nous mesurons l'accélération sur le temps total. Or, le programme global contient beaucoup de code non-parallélisé.
- Pour un nombre de processeurs donné, les problèmes ayant un grand nombre de produits (4, 5 ou 6) obtiennent de meilleures accélérations que les autres.

6.3.2.1.3 Efficacité du tri selon la profondeur

Nous cherchons ici à répondre à la question suivante: si nous éliminons le tri selon la profondeur, quelles performances obtenons-nous? Nous comparons les résultats obtenus avec ceux de l'implantation incorporant le tri selon la profondeur, pour vérifier si celui-ci a un effet positif sur l'exploration parallèle.

Le tableau XIII résume les résultats obtenus en éliminant le tri selon la profondeur. Nous fixons *NPROC* = 4 et *N* = 24. Nous n'incluons comme résultats que le *gap* et *NITER*, puisque nous ne visons qu'à mesurer la qualité des solutions obtenues.

	P1	P2	P3	P4	P5	P6
MZ1	12.0⁺ 898	22.2 193	12.0 308	6.9⁺ 182	6.5 538	3.1 738
MZ2	31.9 843	19.2 966	8.4 130	7.0 321	7.7 40	3.5 706
MZ4	29.4 367	18.7⁺ 871	8.8 57	10.1 64	5.7 35	4.9 266

Tableau XIII. Performances de l'exploration parallèle sans tri selon la profondeur, avec Parallel Fortran
(*NPROC* = 4, *N* = 24)

En comparant les résultats des tableaux XII et XIII, nous remarquons que les solutions obtenues avec le tri selon la profondeur ne sont pas globalement supérieures. Cependant, puisque le nombre de processeurs est très petit, les deux implantations (avec et sans tri selon la profondeur) explorent toutes les deux des portions profondes de l'arbre et fournissent donc des solutions similaires.

6.3.2.2 Express

Tous les tests ont été effectués sur quatre Quadputers, dotés chacun de quatre Transputers. Deux Transputers, dont le Transputer Racine, ont une mémoire de quatre mégaoctets; tous les autres sont dotés d'une mémoire de un mégaoctet. La topologie du réseau d'interconnexion est programmée à l'aide d'un Linkputer. La vitesse des liens de communication est fixée à 20 mégabits/seconde.

Pour tous les essais effectués, la règle du quotient simplifié avec seuil M est utilisée comme règle de branchement. Pour tous les problèmes traités, un maximum de 1000 itérations a été atteint sans que l'arbre ait été complètement exploré.

Nous étudions d'abord la variation des performances en fonction du nombre de processeurs. Par la suite, nous examinons l'effet du tri selon la profondeur en testant une implantation sans tri sur 16 Transputers. Enfin, nous tentons de diminuer le temps des communications, en programmant différentes topologies et en mesurant leur effet sur les performances globales.

6.3.2.2.1 Variation du nombre de processeurs

L'expérience est menée sur une topologie d'arbre binaire balancé, illutrée à la figure 17. Sur cette figure, les numéros des processeurs sont ceux reconnus par le système de communication d'Express (voir section 2.3.2.3). La lettre "H" représente le processeur Hôte (le PC). Le processeur numéroté 0 est le processeur maître, représenté par un "M" sur la figure. Dans cette figure, remarquons que toute topologie utilisant un sous-ensemble de processeurs numérotés de 0 à j , pour $1 \leq j \leq 15$, demeure un arbre binaire balancé. Mentionnons que des tests effectués sur d'autres topologies font l'objet de la section 6.3.2.2.3.

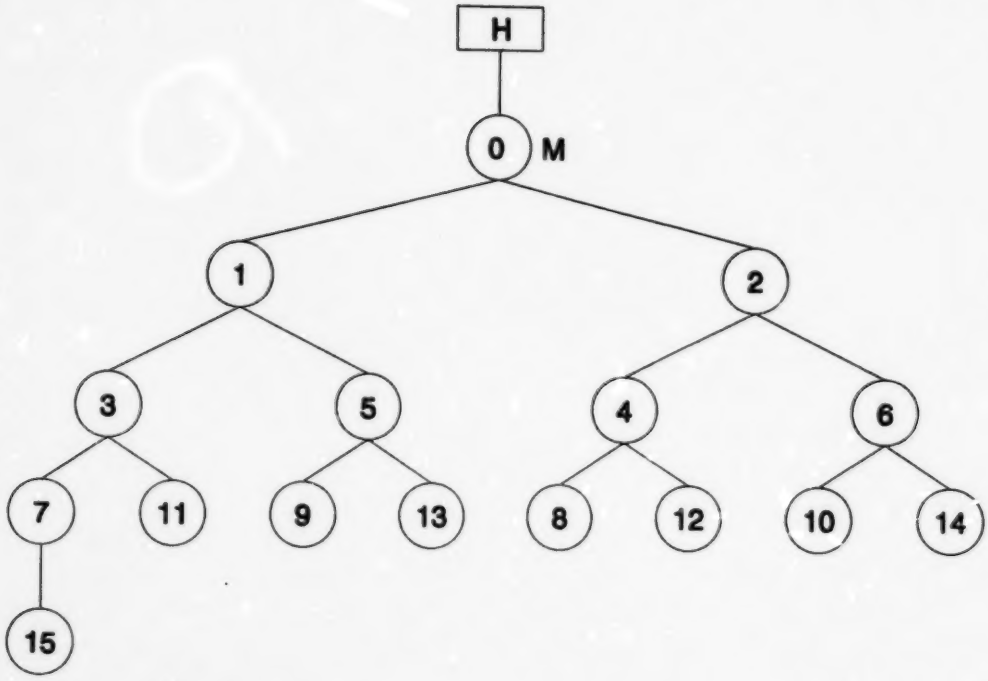

Figure 17. Topologie d'arbre binaire balancé à 16 Transputers

Pour chaque problème, nous fixons $N = 24$ et nous donnons à *NPROC* les valeurs suivantes: 2, 3, 4, 6, 8, 10, 12, 14, 16. Les valeurs 2 et 3 sont ajoutées surtout pour fins de comparaison avec les performances de Parallel Fortran.

Les tableaux XIV à XXII résument les performances obtenues. Pour chaque problème, les tableaux fournissent le *gap*, *NITER* et $a(NPROC)$, l'accélération calculée par rapport au temps total.

	P1	P2	P3	P4	P5	P6
MZ1	12.4 574 1.31	22.2 121 1.31	12.0 411 1.34	7.1 35 1.35	6.5 458 1.37	3.2 29 1.35
MZ2	31.9 357 1.28	19.2 813 1.33	8.4 212 1.33	7.0 554 1.36	7.7 39 1.35	3.5 271 1.37
MZ4	29.4 129 1.32	18.9 686 1.34	8.8 35 1.35	10.1 54 1.36	5.7 42 1.37	4.9 240 1.37

Tableau XIV. Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire
($NPROC = 2, N = 24$)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 400 1.82	22.2 143 1.79	12.0 460 1.86	7.1 38 1.90	6.5 435 1.93	3.1 705 1.92
MZ2	31.9 401 1.76	19.2 667 1.83	8.4 202 1.86	7.0 572 1.89	7.7 43 1.89	3.5 548 1.90
MZ4	29.4 449 1.82	18.9 494 1.85	8.8 36 1.89	10.1 60 1.90	5.7 39 1.92	4.9 240 1.92

Tableau XV. Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire
($NPROC = 3, N = 24$)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 610 2.24	22.2 146 2.23	12.0 415 2.27	7.1 43 2.35	6.5 542 2.39	3.1 743 2.39
MZ2	31.9 319 2.14	19.2 849 2.25	8.4 951 2.25	7.3* 30 2.32	7.7 48 2.35	3.5 418 2.33
MZ4	29.4 398 2.26	18.9 518 2.28	8.8 39 2.35	10.1 71 2.37	5.7 43 2.39	4.9 258 2.40

Tableau XVI. Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire
(*NPROC* = 4, *N* = 24)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 318 2.86	22.2 134 2.85	12.0 274 2.98	7.1 44 3.11	6.5 567 3.19	3.1 924 3.21
MZ2	31.8+ 848 2.69	19.2 721 2.85	8.4 192 2.87	7.0 641 3.07	7.7 63 3.08	3.5 575 3.07
MZ4	29.0+ 944 2.91	18.9 764 2.97	8.8 39 3.08	10.1 67 3.11	5.7 39 3.13	4.9 232 3.18

Tableau XVII. Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire
(*NPROC* = 6, *N* = 24)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 134 3.37	22.2 227 3.33	12.0 457 3.56	7.1 50 3.72	6.5 474 3.80	3.1 776 3.80
MZ2	31.9 454 3.17	19.2 625 3.35	8.4 391 3.41	7.0 641 3.63	7.7 63 3.65	3.5 575 3.65
MZ4	29.4 491 3.46	18.9 762 3.52	8.8 45 3.61	10.1 169 3.69	5.7 50 3.73	4.9 256 3.78

Tableau XVIII. Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire
($NPROC = 8, N = 24$)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 461 3.69	22.2 116 3.71	12.0 321 3.97	7.1 54 4.14	6.5 542 4.27	3.1 795 4.35
MZ2	31.9 396 3.47	19.2 879 3.67	8.4 233 3.74	7.0 642 4.06	7.7 55 4.11	3.5 609 4.08
MZ4	29.4 488 3.81	18.9 475 3.92	8.8 50 4.09	10.1 98 4.13	5.7 54 4.20	4.9 282 4.25

Tableau XIX. Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire
($NPROC = 10, N = 24$)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 127 4.03	22.2 197 3.95	12.0 624 4.29	7.1 58 4.51	6.5 435 4.68	3.1 839 4.70
MZ2	31.8^{+} 890 3.68	19.2 441 3.99	8.4 412 4.05	7.3^{-} 38 4.32	7.7 72 4.33	3.5 819 4.45
MZ4	29.4 592 4.11	18.7^{+} 858 4.31	8.8 41 4.43	10.1 55 4.73	5.7 70 4.59	4.9 353 4.66

Tableau XX. Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire
($NPROC = 12, N = 24$)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 162 4.24	22.2 300 4.20	12.0 748 4.61	7.1 62 4.89	6.5 390 5.01	3.1 955 5.03
MZ2	31.9 305 3.87	19.2 596 4.21	8.4 741 4.23	7.0 691 4.74	7.7 78 4.75	3.5 779 4.78
MZ4	29.4 309 4.35	18.9 496 4.50	8.8 56 4.77	10.1 104 4.86	5.7 73 4.96	4.9 265 5.00

Tableau XXI. Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire
($NPROC = 14, N = 24$)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 387 4.52	22.2 124 4.41	12.0 455 4.79	7.1 66 5.10	6.5 643 5.28	3.1 856 5.38
MZ2	31.9 420 3.97	19.2 949 4.32	8.4 351 4.49	7.3 42 4.87	7.7 87 5.00	3.5 729 5.00
MZ4	29.4 575 4.58	18.9 542 475	8.8 59 4.99	10.1 105 5.13	5.7 66 5.18	4.9 297 5.27

Tableau XXII. Performances de l'exploration parallèle, avec Express sur une topologie d'arbre binaire
(*NPROC* = 16, *N* = 24)

Les mêmes conclusions générales que celles déduites à la section 6.3.2.1.2, s'appliquent ici, soient:

- Quelque soit le nombre de processeurs, les solutions obtenues sont similaires à celles trouvées par l'implantation séquentielle.

- Pour un problème donné, l'accélération augmente constamment en fonction du nombre de processeurs, mais de façon sous-linéaire.

- Pour un nombre de processeurs donné, les problèmes ayant un grand nombre de produits (4, 5 ou 6) ont de meilleures accélérations que les autres.

Ces deux dernières constatations sont illustrées à la figure 18, qui montre les courbes des accélérations obtenues pour les problèmes MZ1P1 et MZ4P6.

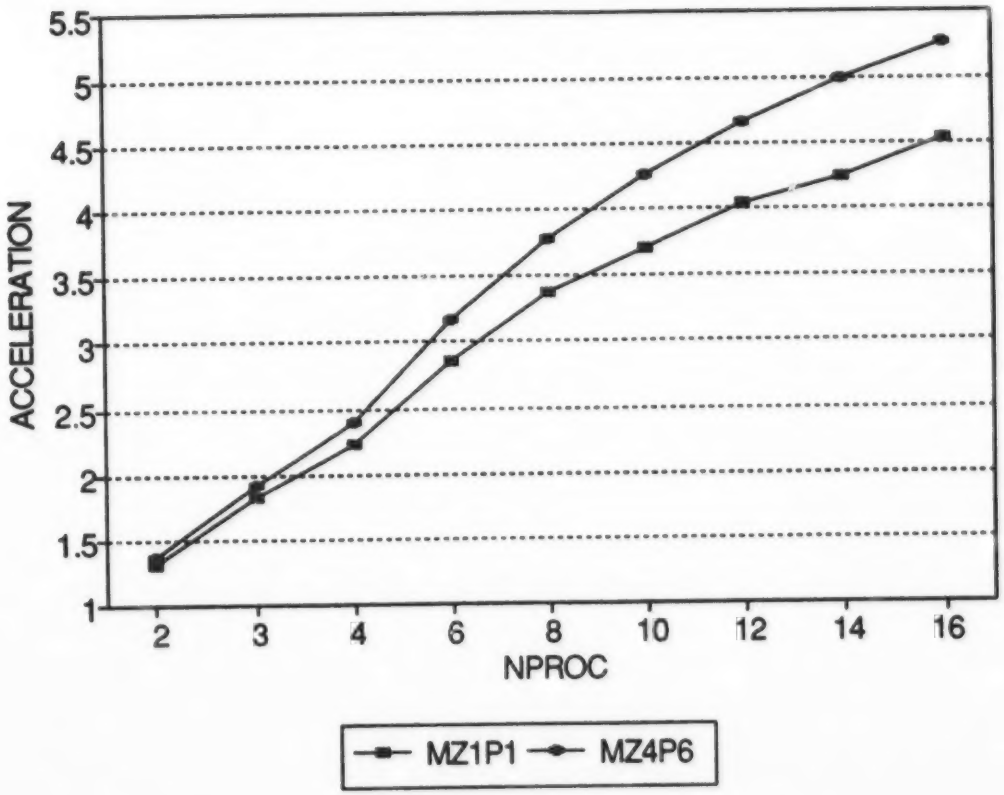

Figure 18. Courbes des accélérations pour les problèmes MZ1P1 et MZ4P6

Si nous comparons les résultats des tableaux X à XII (Parallel Fortran pour *NPROC* = 2,3,4) à ceux des tableaux XIV à XVI (Express pour *NPROC* = 2,3,4), nous remarquons les faits suivants:

- Les deux implantations fournissent des solutions semblables, quelque soit le nombre de processeurs.

- Pour *NPROC* = 2, les solutions et le nombre d'itérations pour les atteindre sont exactement les mêmes pour les deux implantations.

- Pour *NPROC* = 3, les deux implantations donnent des solutions (*gap* et *NITER*) légèrement différentes. Ceci est dû au fait que le processeur maître n'identifie pas nécessairement dans le même ordre les processeurs esclaves avec qui il communique.

- Pour *NPROC* = 2 et *NPROC* = 3, les temps d'exécution sont légèrement meilleurs pour l'implantation avec Express. Ceci s'explique par la vitesse des liens de communication qui est à 10 mégabits/seconde pour les tests effectués sur l'implantation avec Parallel Fortran, et de 20 mégabits/seconde pour les tests avec Express.

- Pour *NPROC* = 4, les temps d'exécution de l'implantation avec Express sont moins bons. Ceci est dû au fait que le processeur numéroté 3 n'est pas directement relié au maître (le processeur numéroté 0) dans l'arbre binaire.

6.3.2.2.2 Efficacité du tri selon la profondeur

Nous testons ici une implantation qui élimine le tri selon la profondeur des problèmes générés mais non-examinés. Nous utilisons 16 Transputers, reliés entre eux par la topologie d'arbre binaire balancé, présentée à la section précédente. Le paramètre *N* est fixé à 24.

Le tableau XXIII résume les résultats obtenus. Pour chaque problème, le tableau fournit les valeurs du *gap* et de *NITER*.

	P1	P2	P3	P4	P5	P6
MZ1	12.5* 680	22.6* 619	12.3* 991	7.1 706	6.8* 114	3.2* 978
MZ2	33.2* 539	19.0+ 460	8.5* 204	7.0 740	7.7 80	3.5 303
MZ4	29.6* 430	19.6* 996	8.8 705	10.1 990	5.7 525	4.9 359

Tableau XXIII. Performances de l'exploration parallèle sans tri selon la profondeur, avec Express sur une topologie d'arbre binaire ($NPROC = 16, N = 24$)

Dans la majorité des cas, l'implantation sans tri selon la profondeur donne de moins bonnes solutions que l'implantation séquentielle. Ceci est dû au fait que beaucoup de temps est passé dans l'examen de noeuds peu susceptibles d'amener rapidement à de "bonnes" solutions réalisables.

De plus, la pile des problèmes générés mais non-examinés croît alors rapidement. Ainsi, il a été nécessaire d'augmenter de 100 à 300 la taille maximale des tableaux contenant l'information de la pile, pour exécuter l'implantation sur tous les problèmes.

Le tri selon la profondeur a donc un effet positif sur les performances de l'exploration parallèle de l'arbre.

6.3.2.2.3 Topologie du réseau d'interconnexion

Notre objectif est ici de construire des topologies qui permettent de minimiser le temps des communications. Comme celles-ci n'ont lieu qu'entre le maître et les esclaves, nous tentons donc d'identifier des topologies qui minimisent la somme des distances du

maître aux esclaves.

Le logiciel Express ne permet cependant pas d'utiliser toutes les topologies qu'on puisse programmer avec le Linkputer. Pour être reconnue, une topologie doit permettre à Express d'effectuer l'allocation des programmes aux Transputers. Cette allocation se fait au moyen d'un mécanisme de distribution des programmes à travers un arbre binaire, ayant comme racine le processeur Hôte. Nous appelons ce mécanisme l'**initialisation**. Ainsi, le processeur Hôte initialise le processeur Racine, puis chaque Transputer ne peut en initialiser que deux autres, au maximum. De plus, chaque Transputer doit initialiser en priorité les Transputers qui lui sont reliés par ordre croissant de numéro de lien. Une autre contrainte pour qu'une topologie soit reconnue par Express, est que le processeur Racine porte le numéro 0.

L'arbre binaire balancé, présenté à la section 6.3.2.2.1, respecte toutes les contraintes imposées par Express. Cependant, la somme des distances du maître aux esclaves, que nous notons D, serait diminuée si nous utilisions un arbre 3-aire balancé. En effet, pour un arbre binaire balancé à 16 noeuds, nous avons $D = 38$, alors que pour un arbre 3-aire balancé, $D = 30$. Le problème est qu'une topologie d'arbre 3-aire n'est pas reconnue par Express puisqu'il n'est pas possible d'initialiser trois Transputers à partir d'un seul. Il faut donc ajouter des liens **artificiels** qui ont pour origine une des feuilles de l'arbre et qui permettent l'initialisation de tous les Transputers. Comme nous l'avons mentionné à la section 2.3.2.3, la communication entre deux processeurs se fait à travers un des chemins les plus courts. Puisque les communications s'effectuent uniquement entre le maître (la racine de l'arbre) et les esclaves (les autres noeuds de l'arbre), les liens artificiels ne sont donc jamais utilisés.

L'arbre 3-aire ainsi construit est représenté à la figure 19. Les liens artificiels sont représentés par des lignes pointillées. Le processeur maître, noté "M", est le processeur 0.

Nous pouvons construire une autre topologie qui permet de diminuer un peu plus la somme des distances entre le maître et les esclaves. En effet, comme nous l'avons mentionné à la section 5.2.1.4, il n'est pas nécessaire que le processeur maître soit le processeur Racine, puisque tous les Transputers ont accès aux E/S. Ainsi, nous pouvons bâtir un arbre dont le maître possède quatre fils et les autres noeuds ont au maximum

trois fils. Le processeur à la racine de l'arbre est numéroté 1. Un de ses fils est le Transputer numéroté 0, qui est relié au processeur Hôte. Le Transputer numéroté 0 peut donc posséder au maximum deux fils. Nous appelons l'arbre ainsi formé arbre "4-3".

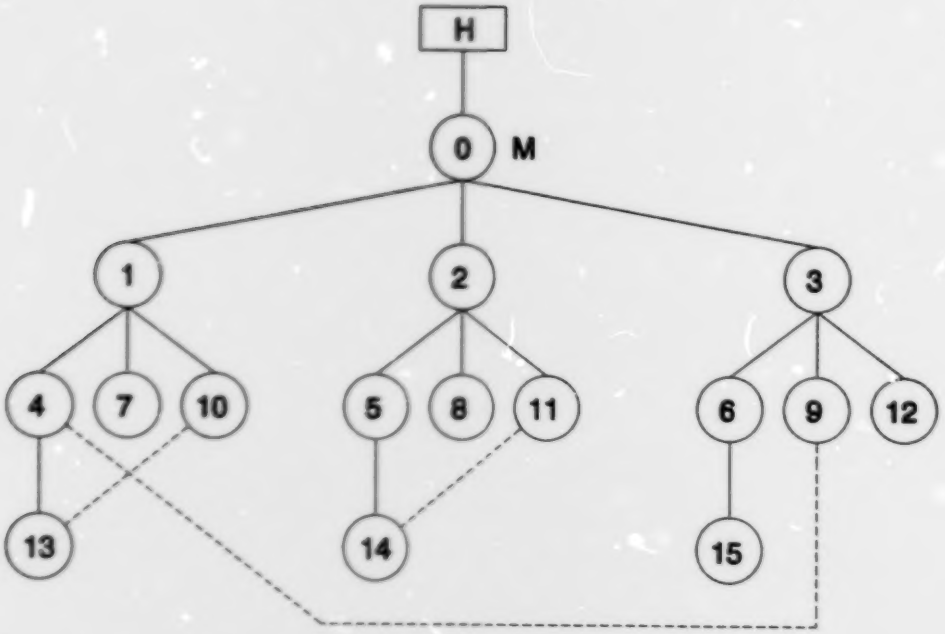

Figure 19. Topologie d'arbre 3-aire à 16 Transputers

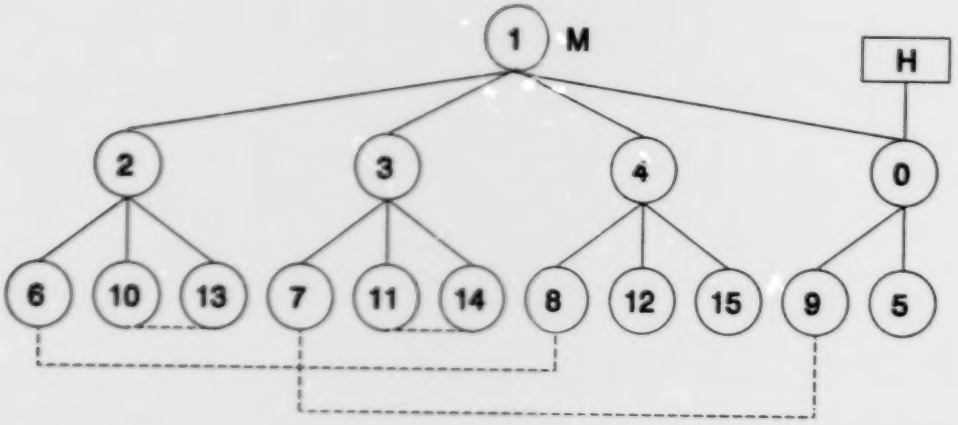

Figure 20. Topologie d'arbre "4-3" à 16 Transputers

La figure 20 illustre l'arbre "4-3", auquel sont ajoutés des liens artificiels représentés par des lignes pointillées. Ces liens sont ajoutés pour permettre l'initialisation des Transputers par Express.

Il est facile de vérifier que pour l'arbre "4-3" à 16 Transputers, la somme des distances du maître aux esclaves est égale à 26. Bien que le temps de communication soit minimisé, il faut prévoir une augmentation du temps des E/S, car le processeur maître n'est pas directement relié au processeur Hôte.

Les tableaux XXIV et XXV résument les résultats obtenus pour les topologies d'arbre 3-aire et d'arbre "4-3" pour $NPROC = 16$ et $N = 24$. Pour chaque problème, les tableaux fournissent le *gap*, *NITER* et $a(NPROC)$, l'accélération mesurée par rapport au temps total.

	P1	P2	P3	P4	P5	P6
MZ1	12.4 382 4.61	22.2 171 4.46	12.0 439 4.94	7.1 949 5.22	6.5 593 5.38	3.2 45 5.42
MZ2	31.9 653 4.07	19.3 506 4.41	8.4 536 4.74	7.3 42 5.07	7.7 88 5.12	3.5 974 5.12
MZ4	29.4 353 4.73	18.9 852 4.88	8.8 44 5.14	10.1 63 5.46	5.7 66 5.28	4.9 314 5.32

Tableau XXIV. Performances de l'exploration parallèle, avec Express sur une topologie d'arbre 3-aire
($NPROC = 16, N = 24$)

	P1	P2	P3	P4	P5	P6
MZ1	12.4 383 4.58	22.2 171 4.44	12.0 483 4.94	7.1* 919 5.22	6.5 603 5.42	3.1 885 5.35
MZ2	31.9 640 4.08	19.2 700 4.51	8.4 506 4.79	7.0 748 5.05	7.7 88 5.13	3.5 397 5.09
MZ4	29.4 346 4.74	18.9 252 4.88	8.8 44 5.17	10.1 63 5.49	5.7 66 5.35	4.9 362 5.38

Tableau XXV. Performances de l'exploration parallèle, avec Express sur une topologie d'arbre "4-3" ($NPROC = 16, N = 24$)

Les résultats montrent que les deux topologies donnent de meilleurs temps d'exécution que la topologie d'arbre binaire. Cependant, il est difficile de discriminer entre les deux. Ceci est dû en partie, au fait que, pour un problème donné, les arbres de séparation et évaluation progressive explorés sont différents, ce qui a un impact imprévisible sur les performances globales.

6.3.3 Calcul parallèle de la relaxation faible

Nous avons effectué des tests concernant le calcul parallèle de la relaxation faible, décrit à la section 5.2.2. Nous utilisons l'implantation avec Parallel Fortran sur un Quadputer à quatre Transputers. La vitesse des liens de communication a été fixée à 10 mégabits/seconde.

Le tableau XXVI résume les résultats obtenus. La première donnée est l'accélération calculée par rapport au temps d'exécution du sous-programme qui résout la relaxation faible à chaque noeud de l'arbre de séparation et évaluation progressive. La deuxième valeur est l'accélération calculée par rapport au temps total.

	P1	P2	P3	P4	P5	P6
MZ1	- -	1.18 1.15	2.00 1.89	2.65 2.43	2.02 1.91	2.10 2.00
MZ2	- -	1.20 1.17	1.80 1.70	2.62 2.40	1.96 1.85	1.96 1.85
MZ3	- -	1.29 1.25	2.01 1.89	2.70 2.49	1.87 1.77	2.22 2.09

Tableau XXVI. Performances du calcul parallèle de la relaxation faible (*NPROC* = 4)

Pour les problèmes à deux ou trois produits, un nombre correspondant de processeurs est utilisé, d'où les accélérations observées. Pour les problèmes à quatre produits ou moins, chaque processeur résout un seul problème de flot à coût minimum. Pour les problèmes à cinq ou six produits, certains processeurs s'occupent de résoudre deux problèmes de flot à coût minimum, alors que les autres en résolvent un seul. Il en résulte un déséquilibre de la charge, qui affecte la performance de l'implantation.

6.4 Application réelle

Nous avons testé trois implantations sur les données de l'application réelle:

- l'implantation séquentielle;
- l'implantation avec Express, pour le calcul parallèle de la relaxation faible;
- l'implantation avec Express, pour l'exploration parallèle de l'arbre du "branch-and-bound".

Dans tous les tests effectués, la règle du quotient simplifiée a été utilisée comme règle de branchement. Pour tous les problèmes testés, un maximum de 1000 itérations a été atteint sans que l'arbre ait été entièrement exploré.

Les implantations parallèles ont été testées sur une topologie d'arbre 3-aire. Nous n'avons pas effectué les tests sur une topologie d'arbre "4-3", car le processeur numéroté 1, qui se trouve à la racine de l'arbre "4-3" possède seulement un mégaoctet de mémoire. Cette quantité était insuffisante pour l'exécution des implantations parallèles sur les données de l'application réelle.

Les performances des trois implantations sont résumées dans le tableau XXVII.

La première partie du tableau ("SEQ") donne les résultats de l'implantation séquentielle. Les données fournies sont, dans l'ordre:

- le *gap*;
- *NITER*;
- le temps total d'exécution moins le temps des E/S;
- le temps total d'exécution.

La deuxième partie du tableau ("PRF") fournit les résultats du calcul parallèle de la relaxation faible. Le tableau donne, dans l'ordre, les valeurs suivantes:

- *NPROC*;
- l'accélération calculée par rapport au temps d'exécution du sous-programme qui calcule la relaxation faible à chaque noeud de l'arbre de séparation et évaluation progressive;
- l'accélération calculée par rapport au temps total d'exécution.

La troisième partie du tableau ("SEP PARALLÈLE") donne les résultats de l'exploration parallèle de l'arbre. Le paramètre *N* prend la valeur 53, qui correspond à l'itération où une première solution réalisable est atteinte. Les données fournies sont, dans l'ordre:

- *NPROC*;
- le *gap*;
- *NITER*;
- l'accélération mesurée par rapport au temps total moins le temps des E/S;
- l'accélération mesurée par rapport au temps total d'exécution.

SEQ	PRF		SEP PARALLÈLE			
	6	12	4	8	12	16
0.8	2.29	2.72	0.8	0.8	0.8	0.8
996	1.96	2.23	993	810	801	866
10079.55			2.29	3.50	4.26	4.74
10740.32			2.15	3.08	3.61	3.92

Tableau XXVII. Performances des implantations sur les données de l'application réelle

Les résultats de l'implantation séquentielle montrent que la relaxation faible fournit une borne inférieure efficace à chaque noeud de l'arbre. Le temps total d'exécution des 1000 itérations est d'environ trois heures.

Lors du calcul parallèle de la relaxation faible, nous avons effectué des essais pour *NPROC* = 6 et *NPROC* = 12. Dans le premier cas, chaque processeur résout deux problèmes de flot à coût minimum alors que dans le deuxième cas, chaque processeur en résout un seul. Pourtant, les accélérations sont très faibles; ceci indique un problème dans le balancement de la charge entre les processeurs. En effet, pour les données de l'application réelle, les réseaux construits pour chaque produit diffèrent sensiblement les uns des autres, ce qui n'était pas le cas pour les problèmes générés aléatoirement.

Les résultats de l'exploration parallèle montrent les faits suivants:

- Les solutions trouvées pour *NPROC* = 8,12,16 sont légèrement moins bonnes que celle trouvée par l'implantation séquentielle, mais comme le montre le *gap*, la différence n'est pas significative.

- L'accélération augmente constamment en fonction du nombre de processeurs.

- Les accélérations sont moins bonnes que celles obtenues par les problèmes générés aléatoirement. Outre le temps accru des E/S, la durée plus grande de la phase d'exploration séquentielle est un facteur pouvant expliquer cette performance.

- Le temps total d'exécution à 16 processeurs est d'environ 45 minutes.

6.5 Conclusion

Les résultats obtenus par l'exploration parallèle de l'arbre de séparation et évaluation progressive, montrent les faits suivants:

- La phase préliminaire d'exploration séquentielle peut avantageusement se poursuivre jusqu'à l'identification d'une première solution réalisable.

- L'accélération augmente constamment en fonction du nombre de processeurs, mais de façon sous-linéaire.

- Pour un nombre de processeurs donné, les accélérations sont meilleures pour les problèmes de grande taille. Cette observation est à nuancer dans le cas de l'application réelle, car alors la durée plus grande de la phase d'exploration séquentielle entraîne une perte d'efficacité dans la performance globale.

- Le tri de la liste des problèmes générés mais non-examinés, selon leur profondeur est efficace.

- Il est préférable d'utiliser le logiciel Express plutôt que le logiciel Parallel Fortran, pour implanter notre algorithme. En effet, Express permet une implantation qui peut s'exécuter sur un nombre arbitraire de processeurs. Quant aux résultats obtenus avec quatre processeurs et moins, ils sont comparables quelque soit le logiciel utilisé.

- Les topologies d'arbre 3-aire et d'arbre "4-3" minimisent le temps des communications. Sur ce plan, l'arbre "4-3" est supérieur, mais augmente légèrement le temps des E/S. Ces observations sont valables pour n'importe quel algorithme de type maître-esclave programmé avec Express.

Le calcul parallèle de la relaxation faible donne de bons résultats lorsque les réseaux construits pour chaque produit sont de tailles comparables (voir les problèmes générés

aléatoirement). Dans le cas de l'application réelle, les réseaux sont de tailles très différentes, ce qui entraîne un mauvais balancement de la charge entre les processeurs et, conséquemment, une perte d'efficacité dans les performances de l'implantation parallèle.

Le calcul parallèle de la relaxation faible peut cependant être utile lorsque combiné avec l'exploration parallèle de l'arbre. En effet, si le nombre de processeurs est élevé, les performances de cette dernière vont plafonner. Il est alors possible d'occuper une fraction des processeurs au calcul parallèle de la relaxation faible. Nous désignons une telle implantation par le terme "hiérarchie d'esclaves", les esclaves s'occupant d'évaluer des noeuds de l'arbre ayant eux-mêmes sous leurs ordres des esclaves chargés du calcul parallèle de la relaxation faible.

CONCLUSION

Le but de ce travail était de réaliser des implantations parallèles, sur des Transputers, d'un algorithme de séparation et évaluation progressive pour résoudre le problème de localisation avec équilibrage.

Pour développer de telles implantations, nous avons d'abord identifié deux sources de parallélisme:

- l'exploration parallèle de l'arbre du séparation et évaluation progressive;
- le calcul parallèle de la borne inférieure à chaque noeud de l'arbre.

Rappelons que dans l'algorithme séquentiel proposé, l'exploration de l'arbre s'effectue par une recherche en profondeur et que la borne inférieure est calculée au moyen d'une formulation de relaxation faible.

Pour l'exploration parallèle de l'arbre, nous avons identifé deux approches principales:

- une approche collégiale, dans laquelle chaque processeur effectue sa propre recherche dans un sous-arbre;
- une approche maître-esclave, dans laquelle un seul processeur, appelé maître, génère les noeuds qui sont, par la suite, évalués en parallèle par l'ensemble des processeurs.

Nous avons adopté cette dernière approche, à partir de laquelle nous avons réalisé plusieurs implantations.

Des résultats obtenus par ces dernières, se dégagent les conclusions suivantes:

- Une implantation parallèle qui explore en priorité des régions profondes de l'arbre fournit des solutions généralement similaires à celles obtenues par l'implantation séquentielle (ceci, lorsque l'arbre n'est pas complètement exploré).

- Une telle implantation est réalisée de manière efficace, par une phase préliminaire d'exploration séquentielle qui se poursuit jusqu'à l'identification d'une première solution réalisable, de même que par un tri de la liste des noeuds générés mais non-examinés, selon la profondeur.

- L'accélération augmente constamment en fonction du nombre de processeurs, mais de façon sous-linéaire.

- Pour un nombre de processeurs donné, les accélérations sont meilleures pour les problèmes de grande taille.

- Les processeurs peuvent être reliés entre eux de manière efficace au moyen de structures d'arbres. Les arbres les plus performants sont ceux pour lesquels chaque noeud a le plus grand nombre de fils possible.

- Pour des données provenant d'une application réelle, l'exploration parallèle à 16 processeurs prend 45 minutes, alors que l'exploration séquentielle requiert 3 heures.

Le calcul parallèle de la relaxation faible repose sur la résolution de plusieurs problèmes de flot à coût minimum mutuellement indépendants. Les processeurs se partagent entre eux les problèmes à résoudre de telle sorte que chacun en ait à peu près le même nombre.

Sur les données de l'application réelle, les problèmes à résoudre sont de tailles très différentes, ce qui entraîne un mauvais balancement de la charge entre les processeurs et, conséquemment, une perte d'efficacité dans les performances de l'implantation parallèle.

Comme suite logique à cette étude, nous pouvons identifier plusieurs avenues de recherche, parmi lesquelles:

- réaliser une implantation qui assure un meilleur balancement de la charge entre les processeurs pour le calcul parallèle de la relaxation faible;

- étudier, concevoir et implanter des algorithmes parallèles pour résoudre un problème de flot à coût minimum;

- fondre l'exploration parallèle de l'arbre et le calcul parallèle de la relaxation faible en une seule implantation; une telle implantation peut être désignée par le terme "hiérarchie d'esclaves", les esclaves s'occupant d'évaluer des noeuds de l'arbre ayant eux-mêmes sous leurs ordres des esclaves chargés du calcul parallèle de la relaxation faible;

- paralléliser la phase préliminaire d'exploration séquentielle;

- identifier et tester des critères auxiliaires, s'ajoutant à la profondeur, pour le tri de la liste des problèmes générés mais non-examinés;

- implanter une approche collégiale;

- calculer la borne inférieure au moyen d'une méthode de résolution basée sur le dual du modèle (Crainic, Delorme et Dejax, 1988); étudier ses effets sur l'exploration parallèle et implanter son calcul en parallèle.

BIBLIOGRAPHIE

3L (1988), *Parallel Fortran User Guide*, 3 L Ltd.

Aikens C.H. (1985), *Facility Location Models for Distribution Planning*, European Journal of Operational Research, vol. 22, pp. 263-279.

Akl S.G. (1989), *The Design and Analysis of Parallel Algorithms*, Prentice-Hall.

Anderson S. et Chen M.C., (1987), *Parallel Branch-and-Bound Algorithms on the Hypercube*, Hypercube Multiprocessor, SIAM, pp. 309-317.

Athas W.C. et Seitz C.L. (1988), *Multicomputers: Message-Passing Concurrent Computers*, IEEE Computer, août, pp. 9-24.

Bertsekas D.P. et Tsitsiklis J.N. (1989), *Parallel and Distributed Computation, Numerical Methods*, Prentice-Hall.

Bhatt S.N. et Ipsen I.C.F. (1985), *How to embed trees in hypercubes*, Department of Computer Science, Research Report YALEU/DCS/RR-443, Yale University, New Haven, Connecticut.

Boxma O.J. et Kindervater G.A.P. (1987), *A Queuing Network Model for Analyzing a Class of Branch and Bound Algorithms on a Master-Slave Architecture*, Department of Operations Research and System Theory, Report OS-R8717, Center of Mathematics and Computer Science, Amsterdam.

Chabini, I. (1990), *Des implantations parallèles de l'algorithme d'approximation linéaire pour la résolution du problème d'affectation du trafic*, mémoire de maîtrise, Département d'informatique et de recherche opérationnelle, Université de Montréal.

Cornuejols G., Nemhauser G.L. et Wolsey L.A. (1990), *The Uncapacitated Facility Location Problem*, Discrete Location Theory, R.L. Francis et P.B. Mirchandini (editors), Wiley-Interscience, pp. 119-168.

Crainic T.G., Dejax P.J. et Delorme L. (1989), *Models for Multimode Multicommodity Location Problems with Interdepot Balancing Requirements*, Annals of Operations Research, vol. 18, pp. 279-302.

Crainic, T.G., Dejax P.J. et Gendreau M. (1990), *Modelling the Container Fleet Management Problem Using a Stochastic Dynamic Approach*, publication no. 685, Centre de recherche sur les transports, Montréal.

Crainic T.G., Delorme L. et Dejax P.J. (1989), *A Branch-and-Bound Approach for the Multicommodity Location Problem with Balancing Requirements*, publication no. 652, Centre de recherche sur les transports, Montréal.

Delorme L., Crainic T.G. et Dejax P.J. (1988), *Dual Methods for Location /Distribution Problems with Balancing Requirements*, ORSA/TIMS Joint National Meeting, Washington, 26-27 avril.

D'Hollander E.H. (1987), *Computer Aided Dataflow Analysis for the Conversion of Sequential Programs into Parallel Form*, Algorithms and Applications on Vector and Parallel Computers, H.J.J. te Riele, Th.J. Dekker et H.A. van der Vorst (editors), North-Holland, pp. 75-102.

Effroymson M.A. et Ray T.L. (1966), *A Branch-Bound Algorithm for Plant Location*, Operations Research, vol. 14, pp. 361-368.

Emmen A. (1987), *A Survey of Vector and Parallel Processors*, Algorithms and Applications of Vector and Parallel Computers, H.J.J. te Riele, Th.J. Dekker et H.A. van der Vorst (editors), North-Holland, pp. 1-45.

Flynn M.J. (1966), *Very High-Speed Computing Systems*, Proceedings of IEEE, vol. 54, pp. 1901-1909.

Flynn M.J. (1972), *Some Computer Organizations and Their Effectiveness*, IEEE Transactions on Computers, vol. C-21, pp. 948-960.

Glover F. (1986), *Future Paths for Integer Programming and Links to Artificial Intelligence*, Computer and Operations Research, vol. 13, pp. 533-549.

Grigoriadis M.D. et Hsu T. (1979), *RNET - The Rutgers Minimum Cost Network Flow Subroutines*, Rutgers University, New Brunswick, New Jersey.

Guo Z. (1990), *Résolution exacte et heuristique du problème de localisation avec équilibrage*, thèse de doctorat, Département de génie industriel, Ecole centrale de Paris.

Hilhorst D., Braam H.D.A. et Sips H.J. (1987), *Some methods for the automatic parallelization of programs; the hyperplane method and its application on the Delft Parallel Processor*, Algorithms and Applications on Vector and Parallel Computers, H.J.J. te Riele, Th.J. Dekker et H.A. van der Vorst (editors), North Holland, pp. 47-73.

Hoare C.A.R. (1985), *Communicating Sequential Processes*, Prentice-Hall.

Hodgson J. (1988), *Application d'une méthode de "branch-and-bound" au problème de localisation avec tranferts entre les dépôts*, mémoire de maîtrise, Département d'informatique et de recherche opérationnelle, Université de Montréal, publication no. 622, Centre de recherche sur les transports, Montréal.

Ibaraki T. (1987), *Enumerative Approaches to Combinatorial Optimization*, Annals of Operations Research, vol. 10-11.

Kaufman L., Eede M.V. et Hansen P., *A Plant and Warehouse Location Problem*, Operational Research Quarterly, vol. 28, pp. 547-554.

Khumawala B.M. (1972), *An Efficient Branch and Bound Algorithm for the Warehouse Location Problem*, Management Science, vol. 18B, pp. 718-731.

Kindervater G.A.P. (1986), *A parallel branch and bound algorithm for the job shop problem*, 8th European Conference on Operational Research, Lisbonne, 15-19 septembre.

Kindervater G.A.P. et Lenstra J.K. (1988), *Parallel Computing and Combinatorial Optimization*, Annals of Operations Research, vol. 14, pp. 245-289.

Kindervater G.A.P. et Trienekens H.W.J.M. (1988), *Experiments with parallel algorithms for combinatorial problems*, European Journal of Operational Research, vol. 33, pp. 65-81.

Krarup J. et Pruzan P.M. (1983), *The Simple Plant Location Problem: Survey and Synthesis*, European Journal of Operational Research, vol. 12, pp. 36-81.

Lai T.-H. et Sahni S. (1984), *Anomalies in Parallel Branch-and-Bound Algorithms*, Communications of the ACM, vol. 27, pp. 594-602.

Lai T.-H. et Sprague A. (1985), *Performance of Parallel Branch-and-Bound Algorithms*, IEEE Transactions on Computers, vol. C-34, pp. 962-964.

Li G.-J. et Wah B.W. (1986), *Coping with Anomalies in Parallel Branch-and-Bound Algorithms*, IEEE Transactions on Computers, vol. C-35, pp. 568-573.

Microway (1990a), *Quadputer Owner's Manual*, Microway Inc.

Microway (1990b), *Linkputer Owner's Manual*, Microway Inc.

Neebe A.W. et Khumawala B.M. (1981), *An Improved Algorithm for the Multi-commodity Location Problem*, Journal of the Operational Research Society, vol. 32, pp. 143-149.

Nemhauser G.L. et Wolsey L.A. (1988), *Integer and Combinatorial Optimization*, Wiley-Interscience.

Parasoft (1990), *Express Fortran*, Parasoft Corporation.

Peyton Jones S.L. (1987), *The Implementation of Functional Programming Languages*, Prentice-Hall.

Pruul E.A. (1975), *Parallel Processing and a Branch-and-Bound Algorithm*, mémoire de maîtrise, Cornell University, Ithaca, New York.

Quinn M.J. (1990), *Analysis and implementation of branch-and-bound algorithms on a hypercube multicomputer*, IEEE Transactions on Computers, vol. 39, pp. 384-387.

Ro H.-B. et Tcha D.-W. (1984), *A branch and bound algorithm for the two-level uncapacitated facility location problem with some side constraints*, European Journal of Operational Research, vol. 18, pp. 349-358.

Tétreault N. (1990), *Une approche de restriction basée sur une méthode primale-duale pour résoudre le problème de localisation-distribution avec échanges interdépôts*, mémoire de maîtrise, Département d'informatique et de recherche opérationnelle, Université de Montréal, publication no. 681, Centre de recherche sur les transports, Montréal.

Trienekens H.W.J.M. (1986), *Parallel Branch and Bound on an MIMD System*, Report 8640/A, Econometric Institute, Erasmus University, Rotterdam.

Wah B.W. et Ma Y.W. (1984), *MANIP - A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems*, IEEE Transactions on Computers, vol. C-33, pp. 377-390.

Zenios S.A. (1989), *Parallel Numerical Optimization: Current Status and an Annotated Bibliography*, ORSA Journal on Computing, vol. 1, pp. 20-43.

REMERCIEMENTS

En terminant, je remercie tous ceux qui ont contribué, de près ou de loin, à la mise en oeuvre de ce mémoire.

J'aimerais d'abord remercier mon directeur, M. Teodor Gabriel Crainic, pour m'avoir proposé ce sujet, pour son soutien constant et pour la confiance qu'il m'a témoignée. Je le remercie plus particulièrement de m'avoir permis de présenter ce sujet de recherche lors de deux congrès scientifiques: à Montréal, aux Journées de l'optimisation, en mai dernier, et à Philadelphie, au congrès ORSA/TIMS, en octobre dernier.

Je remercie également mon co-directeur, M. Michael Florian, pour son rôle déterminant dans la genèse des études sur le parallélisme qui ont vu le jour au Centre de recherche sur les transports.

Je tiens à remercier M. Omar Drissi-Kaïtouni, chercheur au Centre de recherche sur les transports, pour son implication pratique dans le domaine du parallélisme.

Le climat de travail dans lequel a été conçu ce mémoire, n'aurait pas été aussi agréable sans la présence de mon ami Ismaïl Chabini. Je le remercie également pour la lecture des épreuves de ce mémoire.

Ce dernier n'offrirait pas une telle qualité de présentation sans l'aide et les conseils précieux de mon ami Yves D'Astous. Je l'en remercie infiniment.

Je remercie ma soeur, Francine, pour la dactylographie du texte. Elle a accompli son travail avec patience et professionnalisme, malgré les pressions d'un frère parfois exigeant.

Je remercie également le Centre de recherche sur les transports pour l'utilisation de ses ressources informatiques.

En terminant, je remercie le Département d'informatique et de recherche opérationnelle, le Fonds pour la formation des chercheurs et l'aide à la recherche ainsi que le Conseil de recherches en sciences naturelles et en génie du Canada, pour leur soutien financier.